
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática
2008

Sérgio Manuel Silva Freire

Serviço de nomes para TCP/IP

A TCP-layer name service

Departamento de Electrónica,
Universidade de Aveiro Telecomunicações e Informática
2008

Sérgio Manuel Silva Freire

Serviço de nomes para TCP/IP

A TCP-layer name service

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia de Electrónica e Telecomunicações, realizada sob a orientação científica do Professor Doutor André Zúquete, Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Doutor Atílio Manuel da Silva Gameiro
Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Doutor André Ventura da Cruz Marnôto Zúquete
Professor Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro (Orientador)

Doutor Edmundo Heitor da Silva Monteiro
Professor Associado com Agregação do Departamento de Engenharia Informática da Universidade de Coimbra (Arguente)

agradecimentos / acknowledgements

Os agradecimentos são acima de tudo uma oportunidade de relembrar pessoas que os merecem. Entre amigos e família, há acima de tudo duas pessoas que merecem ser realçadas pois sempre desejaram para este que aqui escreve o que não tiveram oportunidade de ter. Estou obviamente a referir-me aos meus pais, Manuel e Arminda. Ainda não foi desta que conseguiram levar o filho ao psiquiatra mas já faltou mais :) Uma menção deve ainda ser feita à PT Inovação, essencialmente nas pessoas dos $Eng.^{os}$ Pedro Salvado, Paulo Nordeste, Marcelino Pousa e à CE, por apoiarem e incentivarem este Mestrado numa perpectiva de valorizacão dos seus recursos...

Fica ainda um especial agradecimento ao meu Orientador pelo apoio e motivação ao longo do trabalho realizado, pela confiança demonstrada na delegação de tarefas, fruto certamente duma mentoria técnica mas sobretudo humana.

Palavras-chave TCP, segurança, resolução de nomes, three-way handshake.

Resumo

A Internet é hoje a maior rede mundial mas para além disso, é também e essencialmente um meio de disponibilização de acesso a conhecimento e a serviços diversos. Tendo como base o protocolo de encaminhamento IP, é possível endereçar e comunicar com pessoas, serviços, máquinas e dispositivos variados. Uma forma de comunicação usual assenta no protocolo TCP, que permite um diálogo bidirecional entre serviços locais e/ou remotos, com tolerância e recuperação face a erros e perda de pacotes. No TCP, um serviço é identificado pelo número do porto a que fica associado, o que tem algumas consequências menos positivas. A mais óbvia é o varrimento de portos (*port scanning*) para posteriores tentativas de ataque a vulnerabilidades nos serviços identificados/associados a esses portos.

Esta tese pretende extender o conceito de endereçamento dum determinado serviço associando-o primordialmente a um nome, ou seja, dotar o TCP dum serviço próprio de resolução de nomes. A fase de estabelecimento da ligação TCP, baseada no *three-way handshake*, pode ser substancialmente evoluída para suportar mecanismos de resolução e de autenticação. A solução encontrada tem a segurança sempre como um aspecto presente e essencial, por forma a combater diversos tipos de ataque.

A resolução de nomes sugerida pode ser integrada com mecanismos de autenticação/validação através do uso de domínios de interpretação (*DOI - domain of interpretation*). Os DOIs possibilitam uma forma flexível de adicionar mecanismos de resolução e autenticação mais ou menos complexos ao próprio estabelecimento da ligação TCP.

Keywords TCP, security, name resolver, name resolution, three-way handshake.

Abstract

Internet is the largest network deployed worldwide but besides that it's also and essentially a way of accessing and distributing knowledge and a way to to interact with services. By using the IP routing protocol it's possible to address and communicate with other persons, services, hosts or network enabled devices. An usual way for establishing a dialogue between internet endpoints is based on the TCP protocol, permitting a bidirectional, reliable and fault-tolerant data exchange. In TCP a service is identified by an associated port number which by itself has some less positive consequences. The obvious one consists on guessing which services are available by find out the available port numbers (*port scanning*) so that attacks on service vulnerabilities can take place.

The purpose of this thesis is to extend the current concept used for addressing TCP services by associating them with names, or simply to provide TCP an in-band name resolution. The connection establishment phase, *three-way handshake*, can be improved in order to support simple name resolution mechanisms or even complex authentication. Security aspects towards avoiding attacks was a major concern that is present in the foundations of the proposed architecture.

The name resolution model can be integrated with several mechanisms for authentication/validation, implemented as logic defined within domains of interpretation (DOI). DOIs allow a flexible and extensible way for adding those mechanisms to the connection establishment procedures of TCP.

Contents

List of Figures

List of Tables

List of Listings

List of Source Code Snippets

Chapter 1

Introduction

1.1 Motivation

Global and distributed attacks, caused directly by attackers or *worms*[1], are very common. One of the main targets of such attacks is the TCP/IP stack and its services which are deployed throughout the Internet and mostly visible to everyone. Even if some firewall rules can restrict access to those services, there is no efficient, transparent and uniform way of imposing a constraint that assures access only to well intended persons or devices.

Historically, some TCP port names are statically bound to well-known services/servers [34]. Examples are `ftp` for port 21, `telnet` for port 23, `http` for port 80, etc. This static mapping between names and ports was initially supported by local services using local data (e.g. file `/etc/services` in Unix systems). Currently there is a database service with all these static mappings [33]. However, these mappings are not mandatory; they just reflect a common use.

Nevertheless, well-known services are coupled to well-known port numbers[2] and attackers are aware of this. As an example, if we know that an email service runs on some host, then we also know it would run on TCP port 25. So, if we want to explore some bug in the email server, we just need to establish a connection to that host, more precisely to port number 25. Port scanners simplify the task of exploring which ports are open on a host; thus, the list of publicly available services running there is easy to obtain.

Given this, other way than numbers should be used to address services. Name systems are useful for translating user-friendly, readable strings into numerical identifiers. A popular name system is DNS [26], used for translating hierarchical, typed names into IP addresses, among other identifiers. Other name services frequently used by applications are RPC name services, such as `rpcbind` for Sun RPC and Microsoft Locator for Microsoft RPC. Until now, there is no widely used, generic name service for transport protocols, such as TCP or UDP.

A simple name service would just provide security by obscurity from attackers sight. Some sort of authentication method, used to validate the process of connection establishment, would impose the first real level of security. There are some mechanisms to achieve this, usually implemented by some out-of-band process like SSTCP and TGTCP [3].

A combined method for in-band name resolution and connection authorization/authentication, before data exchange, seems to be the more obvious and yet efficient way to achieve security, right on the transport layer.

[1] a self-replicating computer program, that uses the network as transport

[2] a TCP port number is for all purposes a TCP address

1.2 Contribution

DNS and RPC name systems are implemented by autonomous servers, using well-known port numbers, which receive resolution requests from the application layer. Their goal is to provide a mapping from a name to a number that could be used as a parameter for lower protocol layers, namely the transport layer. The proposed TCP name system goes in the opposite direction, which is to integrate name resolution into the existing mechanism used for TCP connection establishment: the *three-way handshake*. Consequently, no new or existing name server is used, which is preferable for fault tolerance, and the name service is implemented by the TCP layer.

Besides fault tolerance, the new TCP name service was designed with several other goals in mind. The first one was to maintain compatibility with existing TCP/IP stacks. The second one was to add a new name service, and not to replace the current number-based addressing mechanism used by TCP segments with names, as numbers are more efficient to handle than names. The third one was to allow TCP clients and servers to use arbitrary name formats, in order not to restrict future uses by upper protocol layers. The fourth one was to add arbitrary semantics to name resolution, generalizing the third goal.

Finally, the last goal was concerned with security during the name resolution process. If possible, the model should be robust to attackers that:

- cannot watch the traffic of the remote endpoint;
- could watch the traffic of the remote endpoint;
- could intercept the traffic to the remote endpoint (*man-in-the-middle attacks*).

The first three goals were realized in a simple TCP name resolution model. This model changes slightly the current *three-way handshake* by introducing a resolution request embedded onto the connection request itself. The resolution is made at the remote endpoint and the answer comes on the reply packet that acknowledges the initial request. The model is rather simple but some changes are needed to accomplish this, though the whole model can be implemented on the operating system kernel's TCP stack. The name resolution is actually a name equality match against all listening TCP service names. Although the name itself could eventually contain some form of security aspect (e.g. a one-time password), this was not explored in this specific model.

An extended model was outlined, comprehending all objectives, from name resolution to authentication. The new and enhanced model uses the concept of DOI - *Domain of Interpretation*. The first model's resolution logic is fixed, whereas this version defines a pluggable mechanism that interacts closely with the *three-way handshake*. The interpretation of incoming packets, during the connection establishment, is delegated to DOI Resolvers. Thus, these external user processes can implement the logic they want to either do name resolution, strictly speaking, or do some kind of authentication, or even do both simultaneously. These are the foundations of a generic, secure and extensible TCP name service here presented.

The objective of this work was not to substitute application level authentication, message privacy or integrity assurance. These can be easily provided by already available and standard protocols. TLS [8], as an example, permits those characteristics through a in-band protocol. More, it allows different algorithms to be used for authentication, encryption or for integrity validation, which are negotiated between endpoints before data transactions. But being a interim protocol, between the application and the transport layer, has at least two

consequences: (i) it only goes in action after the connection establishment and (ii) applications must be changed to support TLS. Similarly, some implementations exist directly for the transport protocol, like TCP-MD5 [16] or the recent draft of TCP-AO[3] for integrity validation.

The main focus and purpose of this thesis is the definition of a TCP name service, that:

1. validates the connection during its negotiation phase;
2. is transparent as possible to applications;
3. is extensible;
4. interacts in a backward-compatible way with current TCP stacks.

This dissertation is essentially organized in five chapters, besides this introduction and the ending conclusions.

Chapter 2 covers theoretical concepts focused in networking, with emphasis on TCP details including the *three-way handshake* mechanism. Some protocols and systems for name resolution, their architectures and topologies are also presented. An important subject comprehending not only security but also routing, is depicted when firewalls, protocol scrubbers and NAT are detailed.

In Chapter 3 several worth mentioning, related contributions are analyzed. TCPMUX and DNS SRV records are methods for in-band and out-of band name resolution, respectively. A previously submitted IETF draft has some in-band name-based port addressing, but is not actually a name resolution. Part of this concept was reused for our work but a more general approach was followed instead, overriding some of its limitations. Security and privacy are major issues that can be assured/negotiated right from the start - the connection establishment phase. Port Knocking, single packet authentication, SSTCP, TGTCP, OKTCP are just some solutions more focused on ways of restricting access to TCP services. Secure TCP, as another example, has both security and privacy goals in mind.

The architecture of the proposed solution is described in Chapter 4. In fact, several models are presented, not just one, due to evolution during the time frame of this work. First, a simple name resolution model for TCP is proposed. Later, a complete model involving DOI Resolvers, running in user space, is laid out.

The proposed models were implemented in Linux by means of some kernel hacking in the TCP stack itself. Chapter 5 describes some changes that had to be introduced, some implementation constraints found and some decisions that needed to be taken. API changes, like the one that defines port access restrictions, and new public structures for TCP endpoints are also discussed.

For the purpose of testing the proposed models, some tools had to be developed or adapted. Chapter 6 not only outlines and exposes those tools and developments, but also evaluates the models through conformity (e.g. interaction between current TCP stacks and the proposed modified version) and new feature tests. DOI is also validated by means of some real tests.

The thesis conclusions will summarize the key points of this work, referring benefits and disadvantages. Finally, further study of DOI Resolver models, their direct integration into TCP client/server applications are some of the items enumerated as future work.

The simple resolution model was successfully published as a short paper entitled "A TCP-layer name service for TCP ports"[13] which was presented in June 25$^{\text{th}}$, in Boston during the "Usenix '08 - Annual Technical Conference" and got printed in the conference proceedings.

[3] `http://www.ietf.org/internet-drafts/draft-ietf-tcpm-tcp-auth-opt-00.txt`

Chapter 2

Background

Within this chapter the reader may find some basic concepts on several subjects that served as background for this work. The topics here described cover three different areas: (i) networking and transport protocols, (ii) name resolution and service location and (iii) interferences with the IP end-to-end paradigm.

Special importance should be given to TCP, namely the *three-way handshake* mechanism which was the main focus of the proposed architectural changes. Finally, firewalls and NAT "boxes" are active network components, therefore they can interfere with the communication between two endpoints assuming some standard behaviors, which may no longer be valid.

Name resolution systems or protocols are referred just to have an idea of the different topologies and architectures used for name resolutions.

2.1 Communication through networking

To understand what networking is all about, it is important to clarify some concepts. A **packet** is composed by a group of bits (or bytes), that are joined into a well structured set of information. Packets are exchanged between entities, or **endpoints**, inside **network** environments accordingly with some **protocol** that gives packets their appropriate context. Protocols are usually stacked in a conceptual **network model**, delegating responsibilities and functionalities on each other.

Protocols operating at the same **protocol layer** perform generally the same kind of functions. A **packet flow** indicates the traveling direction for a packet, or group of packets. A connection or **session flow** indicates the initial direction, packet flow, that was used to establish the connection. It might happen that network constraints impose a certain limit for the packet size, also known as the **MTU**[1]. If packets are larger than the MTU, they are either ignored or split in smaller packets using a **fragmentation** method. Network applications communicate with each other using **connection-oriented** or **connection-less** protocols. Their role can be either a consumer/producer of information, which maps a **client/server** model, or a distributed and cooperative worker instead, the case of **P2P**[2] networks.

[1] Maximum Transfer Unit
[2] Peer-to-Peer

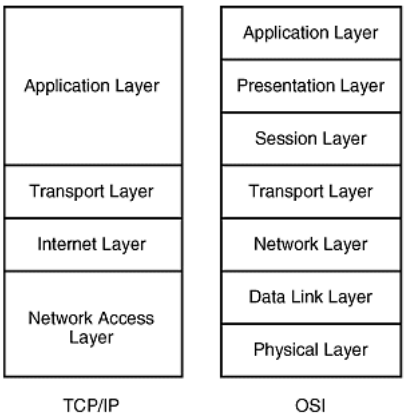

Figure 2.1: TCP/IP model vs. OSI Reference model.

2.1.1 TCP

A **TCP segment** is composed by the TCP header and its data payload. The MSS is the maximum size a TCP segment can have in a specific direction of a TCP stream, excluding the TCP header [29]. The MTU, in the TCP/IP model, defines the maximum size for the IP datagram, which includes the IP header and the TCP segment. While the MTU is imposed by the network, the TCP stack calculates the MSS to fit the MTU and thus minimize fragmentation.

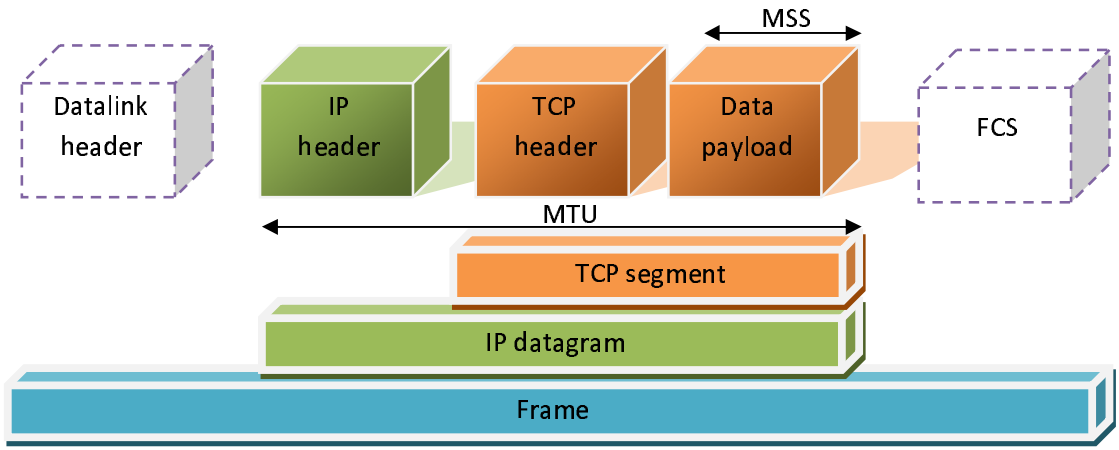

Figure 2.2: Packet encapsulation in datalink, network and transport layers and associated concepts: frame size, MTU and MSS.

TCP[3] [28] is the transport protocol of the TCP/IP protocol stack, on layer 4 of the OSI Reference model. TCP/IP derives its name from the fact that TCP works on top of IP, which is the reason why many applications are said to be TPC/IP rather than just TCP. While a network protocol is used by networking elements for routing purposes, a transport protocol is an interim layer between applications and the network, for appropriate packet delivery to applications within of the destination endpoint. So, it can be considered a sort of

[3]Transmission Control Protocol

application routing with some control mechanisms for providing reliability to end-to-end data transmission. TCP is not a closed standard, it has been and will continue to evolve through time in order to adapt to new circumstances. RFC 4614 [10] provides an overview of TCP, contextualizes it and presents some existing extensions. TCP features include:

- **connection oriented protocol**, reflecting a client/server model, though the standard allows simultaneous opening (a model where both endpoints are clients and servers);
- **full-duplex** and **stream-oriented** communication protocol, since data flows simultaneously in both directions, being delivered in a continuous, streamed way to applications;
- **error detection**, by means of a checksum;
- **error recovery and concealment**, by being able to deal with duplicate or lost segments;
- **reliable data transmission**, through acknowledgments and timeouts using ARQ[4] methods;
- **flow control**, for limiting the rate of data transmitted and thus deal with segment flow constraints;
- **congestion control**, by means of several algorithms [2] that try to avoid network congestion;
- **ordered data delivery** to applications;
- **graceful close**, to guarantee that data still in transit is delivered correctly when closing the connection.

TCP is not the only protocol providing the above features. As an example, the OSI[5] Transport Protocol Layer [17] specified in ISO/IEC 8073:1997, is composed by a suite of five protocols. OSI TP4[6] has some similarities with TCP, more precisely the fact of being a packet oriented protocol for ordered, reliable data delivery supporting full-duplex. Nevertheless, TCP is considered to be simpler (e.g. only one common header format where OSI TP4 has ten) and provides a finer grained control on acknowledged data, since it permits referencing every octet, while the OSI transport protocol deals with data units (blocks of bytes).

TCP segment

A TCP segment is a packet formed by a TCP header followed by data payload. The header size is limited by a 4 bit field (*data offset*) restricting its length to a minimum of 20 bytes and a maximum of 60 bytes.

The **source port** number is a 16 bit field which specifies the originating transport endpoint, that maps to a connection endpoint used by some application to transfer data. The **destination port** number is similar to the previous one, but specifies the destination endpoint. These two fields, while in transit, can be modified (see section 2.3.1) but they are always the ones that will be used for delivering data to the correct services on upper layers.

The header contains two 32 bit sequencing numbers (**sequence** and **acknowledgement**) which are used by the ARQ method. Data within TCP's payload is numbered at octet level, and each flow (client→server and server→client) has its own numbering space, with the initial number defined by the flow initiator. By allowing each data octet to be *indirectly* referred during ARQ, TCP has the basis for a reliable communication protocol.

[4] Automatic Repeat-Query/Automatic Repeat reQuest
[5] International Organization for Standardization, `http://www.iso.org`
[6] OSI Transport Protocol Class 4

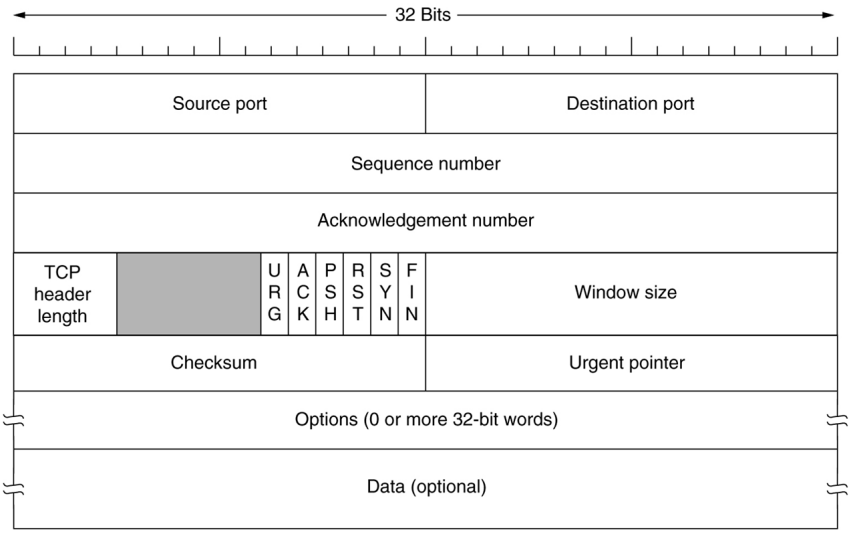

Figure 2.3: TCP header: port numbers identify the connecting points for upper layers while sequencing numbers, window size and checksum provide the means for reliable communications. Data, e.g. from applications, might follow the TCP header.

The header **control bits** are used to signal the support of certain features or to report an event. Initially there were just 6 flag bits (SYN, ACK, PSH, URG, RST, FIN) but latter [31] two bits (CWR, ECE) were taken from the **reserved** field.

Finally, the header contains a variable length field with one or more TCP options. Each option is identified by a *kind* number, for signalling something, or by a *kind* with some specific data attached. These *options* are a way of adding new features without fixedly allocate bits inside the TCP header.

The **sequence number** is the number associated to the first data octet in a segment, which does not apply for packets with the SYN flag active (SYN and SYN+ACK segments). In those cases, its value is the ISN[7]. The **acknowledgement number** is used to acknowledge previously received data (until the acknowledge number minus one), referring the next sequence number the sender is expecting to receive; this meaning is effective only if ACK flag is set. The **window** tells the receiver how many octets, including the one referenced by the acknowledge number, the sender may accept. This has to due with the ARQ method and the sliding window mechanism, described further ahead. As this is a 16 bit wide field, it is rather limited for current transmission throughputs. To accommodate this, RFC 1313 [18] introduced a TCP option named **TCP Window Scale** which scales the window by powers of 2 (each flow has its own scale factor, if supported).

When the URG flag is present, the **urgent pointer** indicates an offset from the sequence number for the boundary of the "urgent data". Data octets bellow the urgent pointer are considered urgent and the receiving TCP stack is responsible for giving this data the proper processing priority, e.g. by adding it to a distinct, high-priority, queue.

A **checksum** is used for basic error detection. Its calculation covers a pseudo header, with network layer addressing information and some transport layer details, the real TCP header (excluding checksum field) and payload.

[7]Initial Sequence Number

TCP Finite State Machine

TCP, like all connection-oriented protocols, has a finite number of states corresponding to a Mealy machine [24] where transitions occur due to systems calls, received packets or even because of timeouts. These states represent the connection life cycle, which involves primarily setting it up, either by initiating it (SYN_SENT state) or else by listening for incoming connections (LISTEN state).

Latter, data transfers take place during the ESTABLISHED state. Finally, either by client's *active close* or by server's *passive close*, the connection is terminated, leading to the CLOSED state, the same one that is used to refer when there is no connection at all.

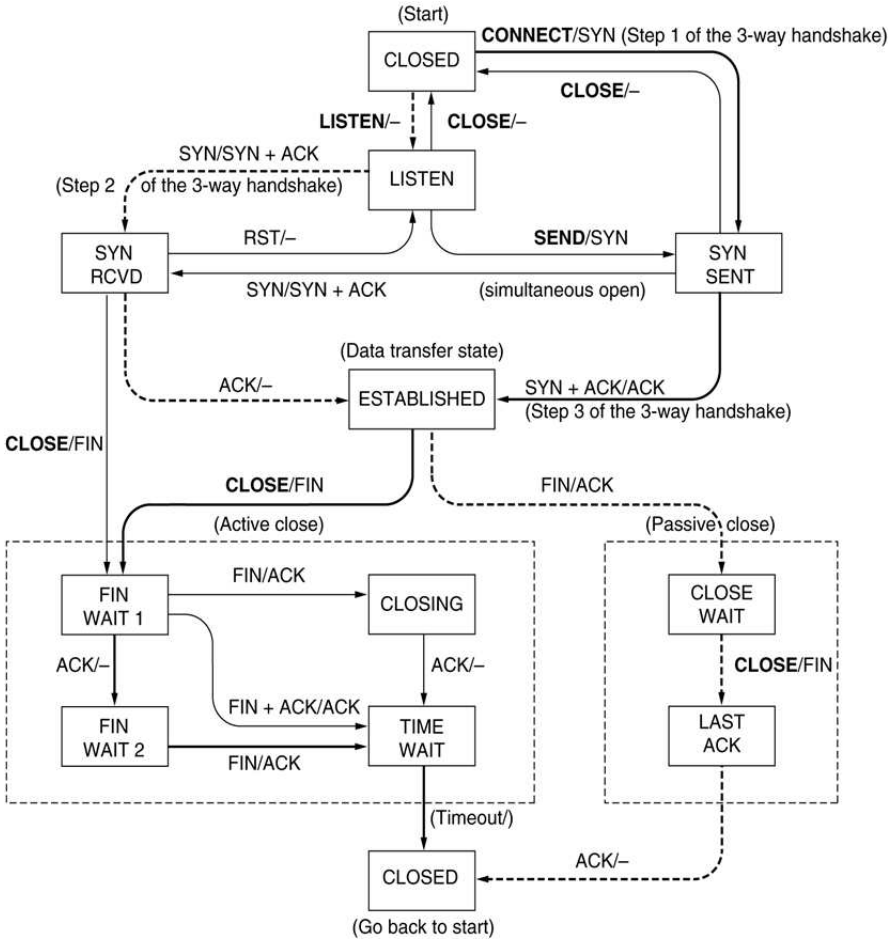

Figure 2.4: TCP Finite State Machine: the typical flow of clients is represented by a continuous dark line; the server typical flow is represented by a dark slashed line. States are represented as rectangular boxes.

When protocols, like TCP, follow a state machine and need to keep track of some connection related variables, then they must reserve some memory for this purpose. A TCB[8] is a structure that must be created at both endpoints to store all needed information related with a ongoing connection.

[8]Transmission Control Block

Three-way Handshake

In TCP, the connection is established using the *three-way handshake* mechanism. Its main purpose it to *synchronize*/exchange sequence numbers that will be used by each endpoint in both packet flows (client→server and server→client). Another purpose of the *three-way handshake* is mutual agreement, that is, a connection is only initiated if both involved endpoints accept the "terms". Some of these are negotiated as TCP options in the SYN and SYN+ACK segments during the handshake. For successful connections, the method works as follows:

1. A client issues an *active connect*. A TCB is instantiated to accommodate the ongoing connection details. An initial sequence number (ISN) for the client→server packet flow is created. Part of this ISN's value is incremented with time while another is randomized, in order to avoid problems with old packets buffered in the network. This ISN is also known as ISS[9]. A SYN flagged segment is sent to the destination, with no other active flag. This packet has no data attached;

2. A listening server (*passive connect*), with its own assigned TCB, receives the SYN segment, does basic packet validation and looks for an application using the destination transport port number. If found, an empty SYN+ACK segment is issued to the originator. This packet has the SYN and ACK flags enabled and the acknowledgement number equals ISS+1. This packet's sequence number is also an ISN, calculated the same way but independently from the received ISS;

3. The client receives the SYN+ACK segment, validates it and then creates an ACK segment with the sequence number ISS+1 and the acknowledge number equal to ISN+1. From the client's point of view, the connection has been established successfully;

4. The server receives the ACK segment, validating it. The server finally considers the connection to be successfully established.

After initiating the connection with the first SYN, the connection process can be aborted through a RST segment sent:

- by the server, instead of the SYN+ACK segment (e.g. if no application is listening on the destination port number);
- by the client, instead of the ACK segment (e.g. if sequence numbers are invalid);
- by the server, when receiving the ACK segment (e.g. if it's not possible to create a connection due to lack of resources).

This RST segment may contain a payload describing the reset cause, though typical TCP stacks do not use this functionality neither provide an API for applications to obtain this. Even if service addressing is not a *three-way handshake* specific feature, it is implicitly associated with it since the transport port numbers are exchanged and used to lookup services during the TCP's connection phase. As mentioned earlier, a server, upon receiving a SYN segment, looks for a listening service endpoint. This is done looking at the destination port number among other fields (e.g. a service can be bound to a TCP port number only on some network addresses). Usually, the client does not specify the source port number, and thus the operating system is responsible for allocating a free, random, port number. Otherwise, the client can indicate a specific port number or reuse an existing one.

[9]Initial Send Sequence number

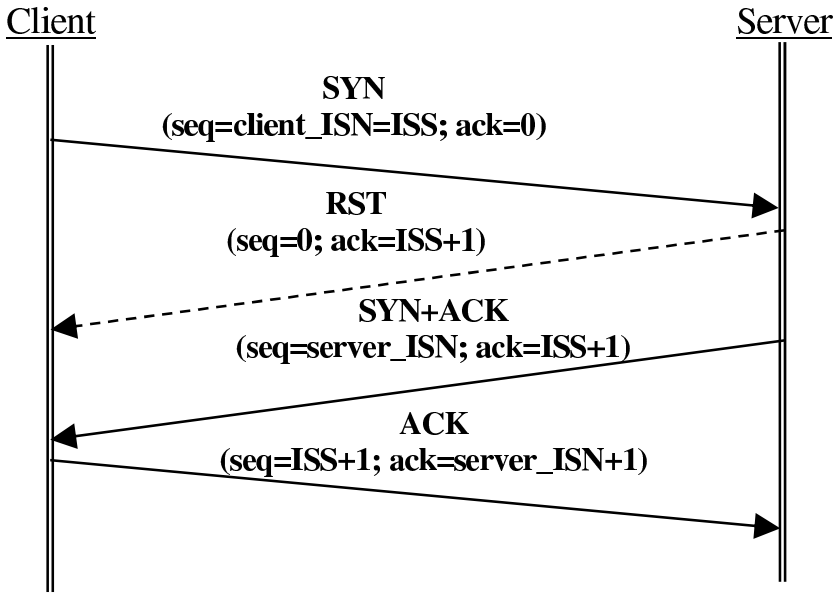

Figure 2.5: Segments exchange in a standard *three-way handshake*, with sequence and acknowledgement numbers. Successful connection flow is represented by filled line while an alternative server reply is represented by a slashed line due to connection rejection (note that RST segment can also occur after receiving SYN+ACK or ACK segments).

A variant of TCP, Transaction/TCP [5] [6] was specified for efficient *request→reply* transactions. T/TCP uses a method named TAO[10] that may fallback to the standard *three-way handshake*. This method combines SYN and FIN flags with data in requests, while on replies a SYN+ACK+FIN segment is sent along with the response. Finally, an ACK is dispatched to the server, acknowledging the received response.

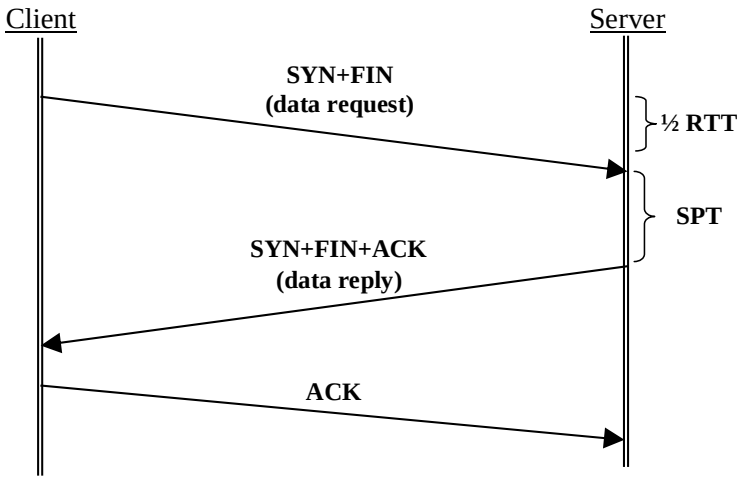

Figure 2.6: Minimal transaction in T/TCP using TCP Accelerated Open. RTT is the round-trip time and SPT is the server processing time.

[10]TCP Accelerated Open

Closing the connection in TCP is not abruptly done. A *graceful close* mechanism ensures that data previously sent is acknowledged by the receiver and that all received data is likewise acknowledged. This mechanism is similar to *three-way handshake* (also known as *modified three-way handshake*) but involves four segments instead of three, since each flow is closed as if it were an independent connection.

TCP Windowing

When a receiver issues a ACK flagged segment, the *window* field informs the destination of how many data octets the receiver is capable of processing - its *receive window*. It also indirectly points to the last acknowledged octet, through the *acknowledgement number*, since acknowledgments are cumulative. In fact, there are two windows used internally on TCP stacks: the *send window* and the *receive window*. The *send window* "informs" the sender of the range of valid sequence numbers acceptable by the receiver. The *receive window* maps a range of valid sequence numbers that the receiver will consider valid. These two windows are implicit concepts of a *sliding window* [7] mechanism, which is used to increase throughput while preserving reliability and flow control. A sender starts by sending several segments, within the *send window*, and a timer starts for each outgoing segment. The timer is cleared upon receiving an ACK segment which explicitly acknowledges a previously received segment or implicitly more, due to cumulative acknowledgment. An acknowledgment can be piggy-backed into a data segment for increased efficiency by avoiding increasing RTT[11]. If it timeouts, the segment is retransmitted. The receiver will issue an ACK when it receives valid data within the *receive window*, acknowledging the last contiguous octet processed. The *send window* "slides" as soon as old octets have been all acknowledged.

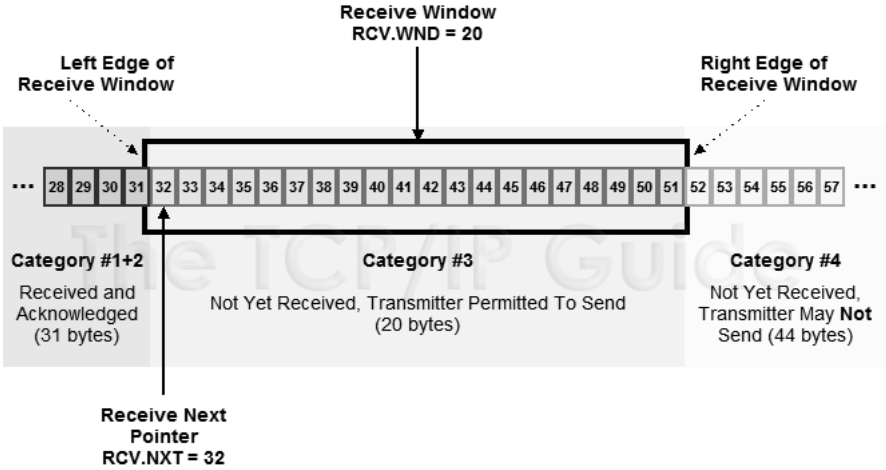

Figure 2.7: TCP *receive window* and pointers to distinguish several categories.

Cumulative acknowledgments behave poorly on lossy or high-bandwidth/large window networks, since any segment can be lost. SACK[12] options [23] introduced the possibility to acknowledge non-contiguous blocks of bytes.

[11] Round-trip Time
[12] Selective Acknowledgements

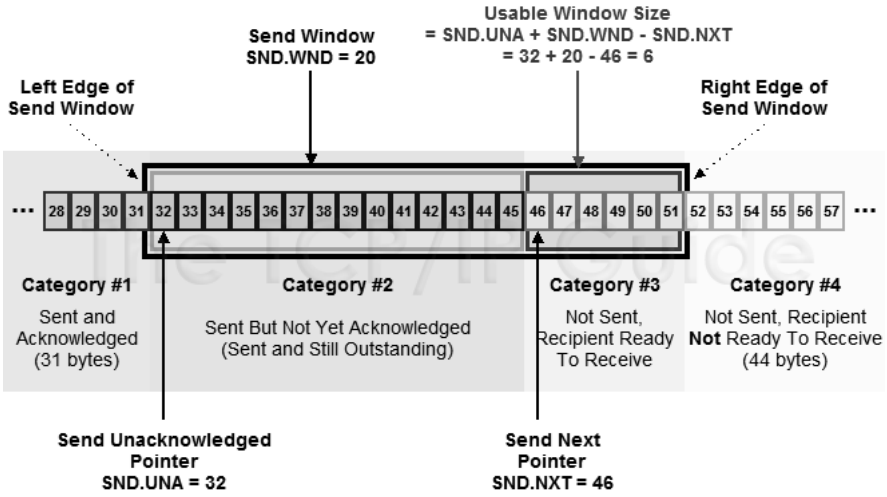

Figure 2.8: TCP *send window* and pointers to distinguish several categories.

PAWS[13] [18] defines a mechanism to protect against colliding sequence numbers due to wrapping of the 32 bit sequence fields on high speed networks. By using a TCP option to embed a timestamp to each segment, it allows the receiver to distinguish old from new packets.

2.2 Name resolution and service location

2.2.1 DNS

DNS[14] [25] [26] is an extensible infrastructure that maps names into addresses. Before DNS, there was an unique file [15] which had all the mappings for every host name. As Internet growth demanded, a distributed, hierarchical name space architecture was devised. The *name database* is distributed, as well as the responsibility to maintain it updated, by several domains and hierarchy dependent DNS zones explained ahead. Each domain is responsible for providing a set of DNS names. A DNS architecture is composed by:

- a **domain name space and RR**[15], which define how information is described and stored in a tree structured way;
- **domain name servers**: daemons responsible for handling name resolution requests. Domain Names under the *authority* of a Domain name server are directly resolved by this server; other names are resolved by other Domain name servers;
- **resolvers**: clients or embedded library functions that interact with name servers for name resolution. These functions are now part of standard libraries, like glibc[16].

Internet names are organized in an hierarchy where names belong to some domain, that may be part of some parent domain and this in turn can belong to one larger domain.

For addressing purposes, this tree like dependency is represented linearly in plain text, interpreted from left to right. Name entries are specified in "." delimited labels of up to 63

[13] Protect Against Wrapped Sequence numbers
[14] Domain Name System
[15] Resource Records
[16] `http://www.gnu.org/software/libc/`

characters each, and can be either described in an absolute or relative manner. An absolute domain ends in "." (e.g. *docs.example.com.*) while a relative name contains just some consecutive starting labels (e.g. using the earlier example, the entry just would be *docs*).

TLD[17] is the topmost domain name of an absolute domain name (e.g. *.com*, *.pt*, ...), which conceptually precedes a final null string - the **root domain**. This special domain is managed by a set of well-known root servers spread around the world. The DNS root is managed by IANA while TLDs are created by IANA/ICANN, which delegates responsibility for each TLD on local organizations.

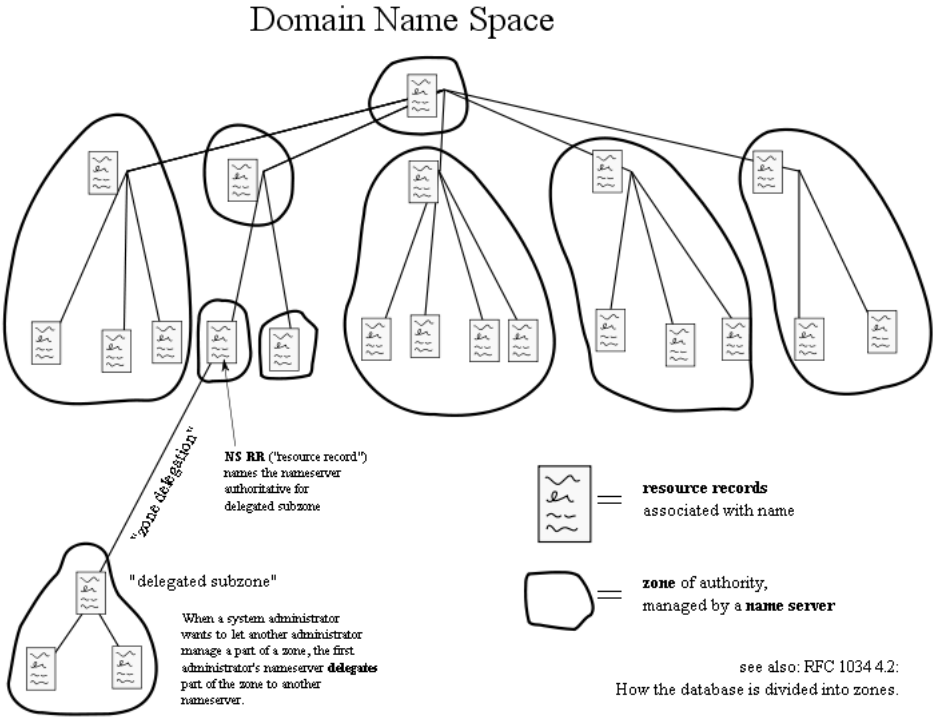

Figure 2.9: DNS and the Domain Namespace.

A **DNS zone** is a subset of the DNS name space, whose management is independent. As an example, an entity responsible for *example.com* domain may want to delegate the *shops.example.com* subdomain name space management to some enterprise. Common RR types are the following: **A**, **AAAA**, **PTR**, **MX** and **CNAME**. An A RR is used to map a host name to an IPv4 address, while an AAAA RR does the same for IPv6. A PTR RR maps an IPv4/6 address to a fully qualified domain name. The MX RR allows the definition of the mail servers responsible for emails sent to mail addresses specifying the domain. By using a CNAME RR, it is possible to create a name alias (e.g. to differentiate services) for existing names. The aliases may be resolved to names outside the DNS zone.

An important characteristic of DNS is that its entries are valid during a TTL[18], which is returned on DNS resolution replies, and can therefore be cached by resolution clients.

A client or library wishing to do name resolution, if not cached, must query the DNS either by issuing a TCP connection to port 53 or by doing a request on top of UDP datagrams, also

[17] Top-Level Domain
[18] Time To Live

to port 53. Queries can be either recursive or iterative. In recursive queries, the first DNS resolver tries to resolve the request if the name belongs to its zone, if not it issues a resolution request to root servers, transversing the tree from up to bottom, until reaching the name server responsible for the zone of the domain. In iterative queries, the client iteratively asks DNS servers for a resolution for the intended name, starting with the local configured DNS server and then going through the root servers, the TLD servers, and down under. Local systems usually cache resolution replies for reducing the overall resolution workload and improving resolution efficiency.

2.2.2 RPC and ONC RPC

RPC[19] is a client/server mechanism used to remotely execute procedures in distributed systems. ONC[20] RPC [38] is an Internet Santdard proposed by Sun that will be hereafter referred as RPC. A RPC program providing some functionality registers itself in a port mapper [37] passing its own program service number, a version number, a local network identifier, an ASCII string formatted (universal) transport address and a string identifying the program's owner. A client that wants to execute a remote procedure on a server contacts the port mapper running at the server's host and asks for the *universal address* of the procedure, which in fact corresponds to the transport protocol details (which one, TCP or UDP, and respective port number) where the pretended service is listening. To invoke the remote procedure, the client contacts the remote server's *universal address*, with the already known RPC program number and version. The port mapper lookup service functionality is in fact a RPC service, having the predefined program number 100000 assigned. It listens in port 111 of either TCP or UDP transport protocols. A port mapper server resolves only local names, i.e. names assigned to local servers.

2.2.3 CORBA

CORBA[21] is a standard mechanism and architecture for allowing communication between objects, independently of their location and of their implementation language. An important component of CORBA is the ORB[22]. This can be an embedded kernel or application function, a set of libraries or even a distinct process. It acts as middleware that makes an access to an object transparent by:

- defining a common interface for object/service management;
- providing location and directory services;
- mediating the communication between objects;

Due to these features, ORB promotes object interoperability. From a client side perspective, the remote object seems local since it is transparently resolved and accessed. Multiple ORBs can communicate with each other, increasing transparency between different domains.

The servers publish the name and the persistent object reference in the ORB while clients only need to obtain, just once, a reference to the object through the name. This is possible because CORBA has a name service interface entitled *CosNaming*[23]. Names associated to

[19] Remote Procedure Call
[20] Open Network Computing
[21] Common Object Request Broker Architecture
[22] Object Request Broker
[23] CORBA Common Object Services Naming

objects are structured in a tree graph and they are resolved in a context, which in this case corresponds to the graph node object and its leaf objects.

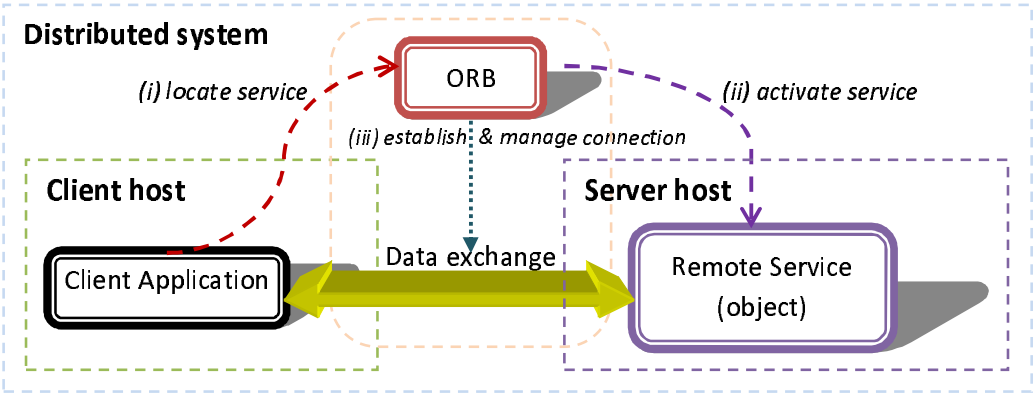

Figure 2.10: ORB - Object Request Broker, acting as a bus and mediating communication.

2.2.4 RMI

RMI[24] is a RPC implementation for Java. In a client/server model or in distributed architectures, objects can exist throughout system nodes. An application wishing to take advantage of this model, must somehow obtain a reference to the object and to the actions (*methods*) it wants to execute on it. In order to locate remote objects and their methods, some steps must be taken though:

- a RMI registry service must be running;
- the remote class must extend a special interface, *UnicastRemoteObject*, and the remote methods must declare to throw a *RemoteException* exception;
- the remote application must bind the instantiated class with the local RMI registry;
- client application must declare the interface used to access the remote methods;
- client application must do a RMI registry lookup on the host where the object exists.

RMI registry name space does not follow an hierarchy concept since any name can be used to bind an object and name spaces are local to hosts. In terms of protocol, applications communicate with the RMI registry and remote objects through TCP ports 1099 and 1098, respectively.

2.2.5 JNDI

JNDI[25] was introduced by Sun, just like RMI, and is an interface, not an implementation, that allows naming and directory services. As name tells, it was specified for Java and its abstraction uses specific Java classes. An *InitialContext*, somehow similar but not equal to a root server in DNS, must be defined. It specifies the underlying mechanism that supports the interaction with JNDI (e.g. LDAP, filesystem, DNS or other). Names are organized in an hierarchy, deriving the fact that a name can be a set of well organized *Names* - a *CompoundName*. Through JNDI naming functionality it's possible to do name lookups, bind/unbind

[24] Remote Method Invocation
[25] Java Naming and Directory Interface - `http://java.sun.com/jndi/`

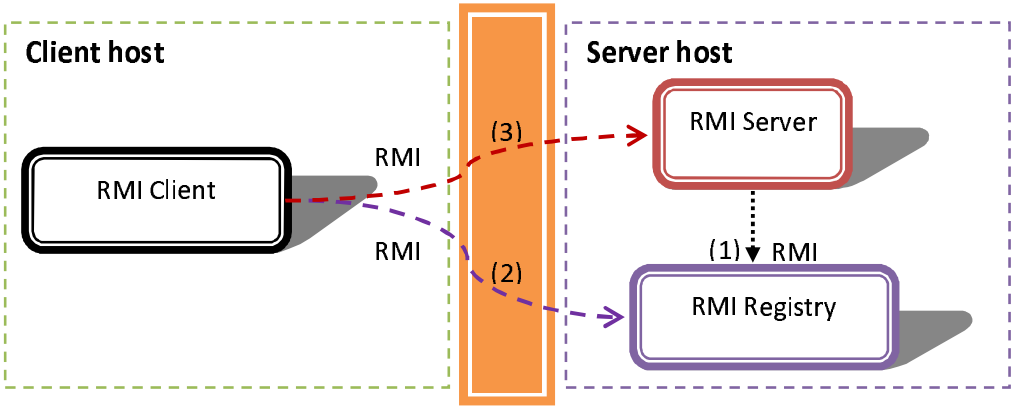

Figure 2.11: RMI: at start a "server"/remote application registers the object in the RMI registry and defines a location for its classes so clients can dynamically download them. A client application that wants to execute a method on a remote object, consults the RMI registry which returns an address to the remote object. If the client does not have the object class definition locally, then fetches it from the remote web server.

names to/from objects or even rename them. The directory interface permits to obtain attributes of certain objects or do searches based on those attributes. There's the possibility to asynchronously receive events when changes occur in naming or directory services.

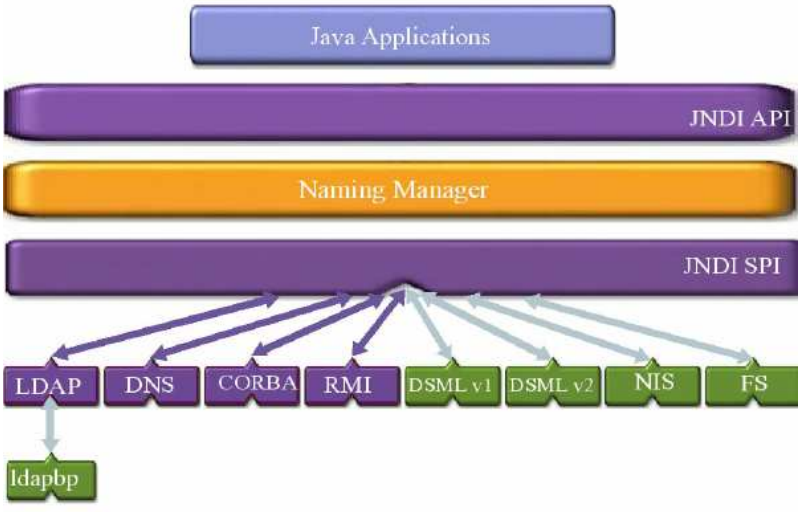

Figure 2.12: JNDI Architecture: multiple underlying implementations for naming services are supported, including RMI.

Applications must use the JNDI API in order to resolve names, which invokes processing logic at the Naming Manager. JNDI SPI[26] is a layer that abstracts the access to underlying naming systems, the supporting base of JNDI.

[26]Service Provider Interface

2.2.6 NIS, NIS+ and LDAP

NIS[27] and NIS+ are mainly directory services, although they can provide naming facilities. The main purpose of these systems was distribution of configuration files across a domain, from a central server(s). NIS+ improved NIS by introducing data authentication and encryption. NIS+ organizes information in a tree structure while the NIS database just allows a plain *key→value* mapping. The underlying protocol is ONC RPC based, meaning that a client must contact the server's port mapper to thereafter communicate directly with the NIS/NIS+ server instance.

LDAP[28] [42] is a protocol [36] for accessing a directory service composed by a organized set of information, through a client/server model. LDAP's DIB[29] contains user and administrative/operational information, which is organized in a tree structure named DIT[30]. Objects governed by the DIT contain a set of attributes, than can be me manipulated or queried. An object is uniquely identified by its DN - *distinguished name*. Although LDAP messages can be directly mapped to TCP[31], any connection-orient transport protocol can be used.

2.3 Interferences with the IP end-to-end paradigm

2.3.1 NAT and PAT

A widely deployed routing solution is NAT[32] [12]. NAT was originally devised in order to solve two problems: (i) IP address depletion and (ii) scaling in routing. While the first could be achieved by newer protocols like IPv6, the latter is not so easy to carry out. By means of address reusing, NAT allows hosts of different realms to transparently communicate with each other. The primary key assumption for NAT is that only a very limited number of hosts, at a certain time, are in fact communicating to hosts outside of the domain. Considering this, only very few addresses might be needed from the outer of domain perspective.

Every realm contains its address space that can collide with some other domain's address space. NAT operates at the borders of the domain, acting on inbound and outbound traffic. An entity doing NAT must translate inner domain addresses into reusable outer addresses, which are either a subset of the domain's original address space or simply belong to another network range. Considering one outgoing connection, to the domain's exterior, the packet's source IP is translated to one free reusable address at the NAT entity. Therefore, during the session existence, a mapping table must exist for a correct packet routing, for either packet flow.

A side effect of using NAT is increased privacy because the private addresses of the domain are not seen in the exterior; they are translated to a set of global, or even public, addresses. Since some applications and upper protocols negotiate addresses and ports inside the packet's or protocols' payload, ALG[35], as defined in RFC 2663 [39], need to couple and interact with NAT in order to produce a correct and complete translation. FTP and SIP are good examples of protocols that need specific ALG logic for NAT transversal.

[27] Network Information Service, sometimes referred as Sun's *Yellow Pages*
[28] Lightweight Directory Access Protocol
[29] Directory Information Base
[30] Directory Information Tree
[31] Usually LDAP uses TCP ports 389 or 636 (SSL). Microsoft's Active Directory, uses 3268 and 3269 instead.
[32] Network Address Translator
[35] Application Level Gateway

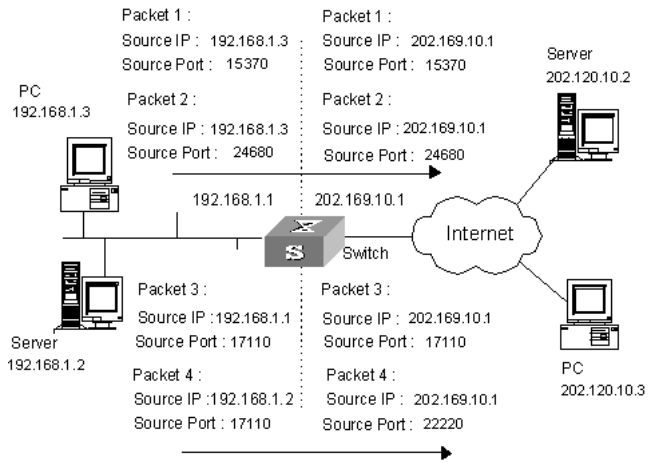

Figure 2.13: NAT and PAT (`http://www.h3c.com`): packets 1, 2 and 3 are translated through NAT from 192.168.1.0/24 network to the switch's public IP 202.169.10.1, while packet 4 source port is translated from 17110 to 22220 through PAT[34].

Major advantages can be summed as:

- Address reusing, providing domain administrators an efficient way for solving the address space limitations;
- Multiple, cascade NAT operations can be performed transparently for most applications;
- Apparent privacy for domain hosts, since their IP is not advertised outside of the domain. A more important drawback, related with this, is defined bellow.

Although its apparent simplicity, some disadvantages may be pointed to NAT.

- The connection semantics from end-to-end is somehow lost, because IP's are translated in every hop that performs NAT. Moreover, there's *distributed state information* that must be maintained during the session, so all packets are routed correctly (the packet by itself no longer has all the necessary routing information). Besides memory usage, the NAT entity must also keep track of session flows and know exactly when to remove idle *sessions*. Given this, DoS attacks can arise.

- Applications must be NAT aware if they use, at protocol layer, IP addresses or transport ports to do some kind of validation or for processing reasons. This application layer modification might not be trivial.

- Problems dealing with current, NAT incompatible, deployed protocols. An entity performing NAT, must be application level compatible through some ALG, that means that it has to know upper layer protocols and modify packets accordingly. Besides changing IP address and updating checksum on the packet, it might need to modify also the TCP header (checksum, sequence numbers) or even the TCP payload.

- Constraints on security and privacy mechanisms. As layer 3 (IP) has to be modified and layer 4 (TCP/UDP) possibly has to be adapted, information contained on these layer's headers cannot be used safely for some security mechanism based on IP's, TCP header checksums, etc. Also, it can be hard to track the real origin of a problematic traffic source since the packet's source IP is no longer the original one.

Nowadays, an evolution of the original concept behind NAT is used to map either pairs of ⟨*src IP*, *UDP/TCP src port*⟩ or ⟨*dst IP*, *UDP/TCP dst port*⟩ for outgoing or incoming packet flows, in a certain domain, respectively. Entities or devices using NAPT[36] [39] must keep a table with these pairs. This mechanism increases address reuse because very few external addresses need to be mapped to, since UDP/TCP ports are also used for the address translation and routing. There are also several NAT variants like *Twice NAT* that does NAT for the source and destination addresses (IP + UDP/TCP port).

2.3.2 Protocol Scrubbers

Networks are very heterogonous, containing a mixture of devices and hosts with different stack and applicational behaviors. Attackers wishing to take control of systems can take advantage of the different singularities of each implementation inside the domain. Practical examples include, insertion and evasion attacks [30] which are used to modify traffic seen by a certain peer, in this case an IDS[37]. This is accomplished by inserting specific packets in the network which are accepted by the IDS but rejected by the destination peer - *insertion attack* - and also packets that are ignored/rejected by the IDS but that are in fact accepted by their destination - *evasion attack*.

A protocol scrubber [22] is an active, transparent mechanism used for traffic normalization. It can act either at transport or at application levels. A protocol scrubber defines a set of common rules to validate a specific protocol and may impose a common flow accordingly with a state diagram, where applicable. Besides shaping traffic and consequently constraining attacks, protocol scrubbers can also behave like active NID[38] systems, reporting whenever some endpoint tries to explore some protocol *hidden feature* or some implementation characteristic. By eliminating protocol ambiguity, outside attackers only see one interpretation of that protocol and are forced to deal with more restrictive rules which are normally not explicitly specified by standards. An example can be a TCP/IP protocol scrubber (see Fig. 2.14) where TCP fields can be validated more restrictively and certain values can be either forced or denied. Also, the way overlapping or duplicated segments are processed is standardized by means of this mechanism.

2.3.3 Firewalls

A **firewall** is a network security barrier. There are several definitions for firewalls due to recent evolution on two aspects: networking and security. A possible definition for a firewall could be: "**firewall - a functionality provided by software or hardware device(s) that allows, restricts, inspects, modifies, encrypts, tunnels or proxies network traffic between different security domains accordingly with some rules**". Nowadays, most firewalls have built-in NAT/PAT support. It is also becoming common to find VPN functionality among the firewall features.

In its more simple and earlier version, a firewall just did basic packet filtering, operating at the network and/or transport layers. As every packet was treated independently, these firewalls were called to be *stateless* since no extra information was necessary for the decision logic. Most traffic is bidirectional, thus rules had to take this into consideration, so besides

[36] Network Address Port Translation
[37] Intrusion Detection System
[38] Network Intrusion Detection

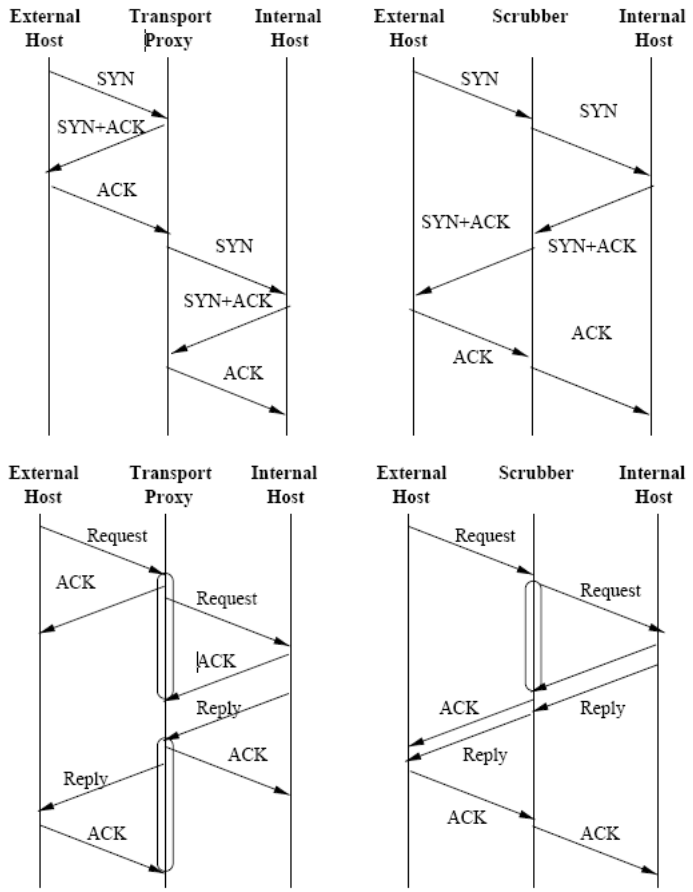

Figure 2.14: TCP protocol scrubber: unlike an ordinary transport proxy, a protocol scrubber acts actively on traffic, validating and enforcing common rules. A transport proxy relays traffic.

rules *n*'plication, there was a huge limitation preventing an accurate way of defining correct and fully restrictive rules.

A demand for connection aware firewalls emerged and *stateful firewalls* started to appear. The state can be just a pair of address related values or have even more complex information, like the local/remote mapping used in NAT or other used by VPN.

A simple stateful firewall implementation is a circuit-level firewall that monitors the connection handshake and only forwards packets if a connection was previously authorized. A connection identifier (like source and destination IP addresses and TCP/UDP ports), sequence numbers data, among other, must be saved in the state structure.

An application layer firewall goes a step further by analyzing data exchanged in packets, typically from the application layer. FTP or SIP are examples of protocols that negotiate transport protocol details at application level.

Application gateways are proxies that relay traffic for a well known protocol, which is used by some application. During transactions, two connections exist: one between the local "client" and the proxy and another one between the proxy and the remote endpoint. There is no transparency for clients because they must be proxy aware.

Dynamic packet filtering or stateful inspection, is a firewall method that looks deeply at the packet, analyzing the network layer up to the application layer, so packets are fully evaluated in their full context. This method works transparently, unlike application gateways.

There are tools for Linux based distributions that permit IPv4 and IPv6 packet filtering and NAT/PAT based on an ordered list of rules: **iptables** and **ip6tables**. Because Linux permits connection tracking through the *netfilter* framework[39], it's possible for iptables to behave like a stateful firewall.

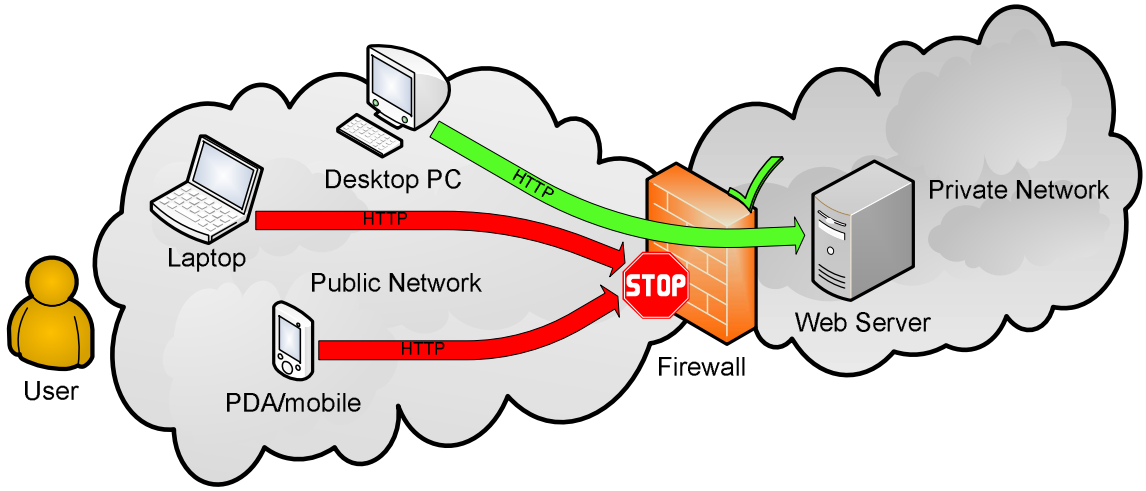

Figure 2.15: A firewall in action: HTTP requests to the Web Server are only allowed when originated in the Desktop PC. The firewall rules only allow traffic for the destination TCP port 80 coming from the Desktop PC's IP address.

[39] `http://conntrack-tools.netfilter.org`

Chapter 3

Related Work

In this chapter are briefly described several contributions concerning two subjects: binding of names to transport ports, or services, and access control mechanisms.

3.1 TCPMUX

TCPMUX is a service using TCP port 1 which allows a host to provide a port name handoff service for itself [21]. A client host opens a connection to port 1 on a server host and transmits the desired port name in the data stream; the server replies with a positive or negative name resolution by means of a reply character in the data stream. If the named service is available, the connection is transferred to the desired service.

In Linux systems, the TCPMUX service is provided for named services handled by the *inetd* daemon. Thus, arbitrary servers cannot provide name-number bindings to TCPMUX; only servers listed in *inetd* configuration files can have named ports.

The use of TCPMUX is not transparent to clients, as they must use in a different way the connection to a server: first they must contact the TCPMUX and provide the port name, then they interpret the TCPMUX reply and only afterwards, in case of success, they may proceed with the intended client-server interaction. Most client-server applications using TCP use a different approach, they contact directly a target server using its port number.

3.2 DNS SRV records

RFC 2782 [14] defined a new type of DNS records, SRV RR, for resolving port names to a set of ⟨*host DNS name*, *port number*⟩ pairs. By creating *_service._proto.name* SRV RR entries in the DNS, domain administrators are able to specify a set of locations (hosts) of a given service described by a friendly name, for the domain. For example, when the name `_foobar._tcp` is resolved in a DNS domain, it returns a set of ⟨*host DNS name*, *port number*⟩ pairs where the `foobar` service over TCP may be found.

These records are very useful for locating public services with well-known names (e.g. SMTP, FTP servers) in a domain, but are not well suited for dealing with arbitrary port names used in ad-hoc client-server connections, because:

- The client application must first learn the DNS domain of the target host before resolving the port name. For instance, to connect to port `foobar` at host 192.168.1.1, the client

must first discover the DNS domain of 192.168.1.1 (e.g. `example.org`) and then resolve the name `_foobar._tcp.example.org`;

- The server host must have a DNS name so that clients could match target IP addresses with host names returned by SRV RR resolutions;
- Requires frequent DNS updates, and the inherent administrative privileges to do so, in order to dynamically create SRV RR entries whenever servers bind names to port numbers. This is a major blocker for dynamic port name definition and for normal end-user usage of port names.

3.3 Port Knocking and Single Packet Authorization

Port Knocking[1] (PK) is a mechanism for restricting access to sensitive services, by allowing only authenticated requests to reach a server. It is a passive authentication scheme for TCP connections, acting as an auxiliary and external mechanism, independent of the kernel and the applications. The so called *knock* or *authentication*, as originally devised, is a sequence of connection attempts to closed ports, which are intercepted and validated by a Port Knocking daemon at the server host (or some intermediary firewall). After a correct *knock* by a client, the daemon allows it to connect to the wanted service port.

The sequence of ports contacted in a *knock* can have an encoded meaning, like the origin IP, the remote port number and a checksum, that allows further control of the connection establishment. By introducing a random number as a one time password, it's possible to limit replay attacks.

There are different flavors of Port Knocking mechanisms, though. As transport protocols, either UDP or TCP SYN packets can be used to provide the authentication information. While the former might seem more appropriate, mainly due to its connection-less characteristic, the latter has a bigger header with some extra fields, besides the source/destination port numbers, that can be safely used to encode additional authentication information. Beyond TCP and UDP, ICMP can also be used as a "transport protocol" for the authentication data (implementations include Barricade[2], Cerberus[3]). In this case, an ICMP echo packet, "*ping*", is crafted with a one time password which is in fact a hash of variables specific to the connection. In Single Packet Authorization [32] (SPA), a deviation of PK, a single packet contains the necessary information to authorize the TCP connection. SPA aims to present an alternative solution to the original PK that used just TCP/UDP port numbers to encode the authentication information. Combining all in the data part of just one packet increases efficiency and throughput, while allowing additional security features to be built-in due to the available space within the packet.

Port Knocking is a simple concept that can be easily implemented using already available tools. Nevertheless, its implementation requires:

- a Port Knocking daemon between the client and the server;
- a firewall between the client and the server and a mechanism to interact with it in order to modify permission rules as needed;

[1] `http://www.portknocking.org`
[2] `http://www.lightning.eu.org/barricade/`
[3] `http://silverstr.ufies.org/blog/archives/000625.html`

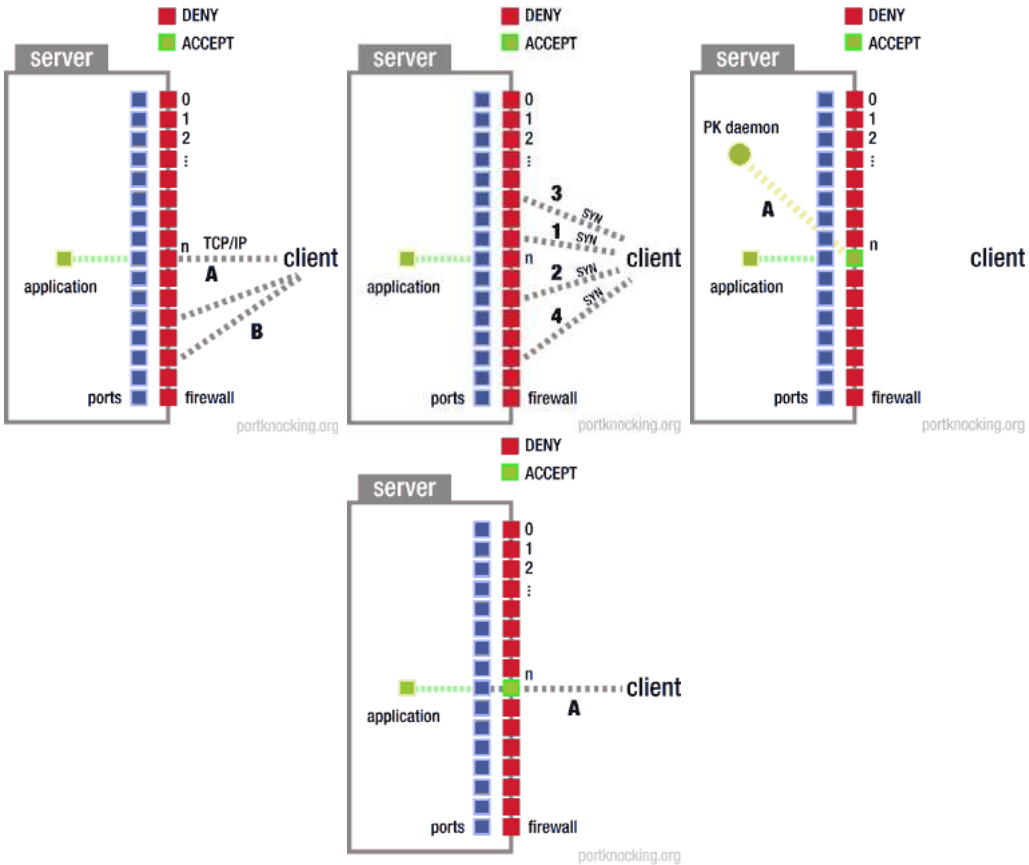

Figure 3.1: Original conceived PK mechanism, explained from left to right: client wants to connect to the service available through TCP port n but that port, like all others, is closed from outside access. Client then issues a series of connection attempts, in a specific order, that will be processed by a port knocking daemon. The latter validates this sequence and, if successful, opens a firewall rule allowing the incoming connection to the service listening on TCP port n.

- a range of closed ports for detecting knocking sequences.
- a client application, or some library functions, to carry on the *knock* before starting a connection to a protected port.

Some problems also arise from PK usage, such as the validation of source IP's in networks where NAT is performed. Some workarounds (like **fwknop**[4]) exist but they tend to increase the complexity of the solution.

3.4 SSTCP, TGTCP and OKTCP

A paper [3] published by Intel introduced some interesting techniques focused on TCP services concealment. The three proposed approaches were depicted for accessing and securing TCP services either by changing the connection philosophy (e.g. involve additional packets

[4] `http://www.cipherdyne.org/fwknop/`

for authorization/authentication) or by slightly extend the *three-way handshake*. The main objectives included: security by obscurity, lightweight authorization and end-to-end authentication. A description of the three variants follows.

- **SSTCP**, or Spread-Spectrum TCP, makes use of several TCP SYN segments targeted for the remote peer, that must be sent before doing the standard *three-way handshake*. Within each SYN packet, one specific TCP header field (e.g. ISN or destination port number) is used to encode a key that is validated at the remote endpoint side. The name *spread-spectrum* is a metaphor: in telecommunications a signal is spread trough several frequencies (e.g. the frequency spectrum), likewise in SSTCP the key is "spread" throughout several TCP SYN segments. This is in fact very similar to the Port Knocking mechanism, if the destination TCP port number is used to encode the key. As a drawback, it can be pointed the fact the key is a previous agreed shared key between both peers. The amount of SYN segments used to encode the authorization key must also be known by the participating entities beforehand.

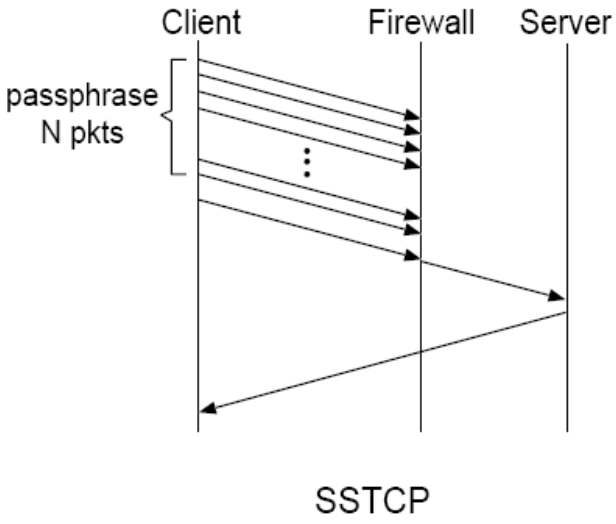

Figure 3.2: Spread-Spectrum TCP

- **TGTCP**, or Tailgate TCP, needs an additional packet before the *three-way handshake* passes through. That initial packet can be either an UDP datagram or a TCP SYN segment and it contains a shared key and other data to authorize the connection that follows. Furthermore, there is also some timing information to restrict replay attacks. An important drawback can arise from synchronization issues, that is, the authorization packet can arrive either too early or just too late due to network routing.

- **OKTCP**, or Option-Keyed TCP, does not need an additional packet to provide the necessary information to be used for authorization purposes. Instead, it uses either an IP option (i.e. *IP timestamp*) or a TCP option (i.e. *TCP echo*) instead. Both solutions have one major restriction: the space available to encode a key is very limited.

It must be mentioned that authors point out that the data part of a TCP SYN segment could eventually be used to transport the key information, either in TGTCP or in OKTCP.

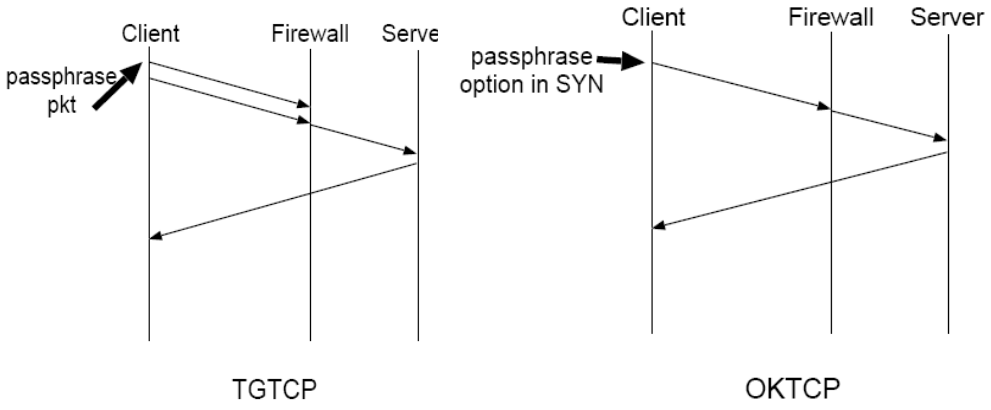

Figure 3.3: Tailgate TCP (left) and Option-Keyed TCP (right)

3.5 Secure TCP

With the goal of adding security and increased integrity directly to the transport layer, a mechanism [40] was conceived long time ago for this purpose. Secure TCP, as it was called then, works by extending the *three-way handshake* as seen in Fig. 3.4.

In a first phase, the client sends a SYN segment with its own capabilities in terms of integrity and confidentiality. Later on, the server answers with an "enhanced" SYN+ACK segment, where it identifies a compatible encryption method that should be used on the following segments along with the server's public key. The client then replies with its own public key and session shared keys encrypted with the provided server's public key. One shared key is used for integrity while the other one is used for TCP segment encryption during the normal data exchange, after the connection establishment. The TCP header itself keeps unprotected for backward compatibility but the TCP body tail contains a MAC (Message Authentication Code) that allows integrity validation comprising the TCP header and the TCP encrypted data inside the TCP body part.

Using this solution has its own pitfalls, namely:

- implies that both peers must have valid certificates, which should be signed by the CA (Certificate Authority) for all IP addresses used by peers on "secure TCP" connections;
- assigns a bit named "TCP options extension" to one of the bits in *reserved field* of TCP;
- introduces 5 new TCP options, namely "TCP length", "negotiation send", "negotiation reply", "key exchange send" and "key exchange reply", although the last four ones are contained within the TCP data part;
- inhibits the use of cross-active open;
- implicitly implies an additional ACK segment at the end of the connection establishment;

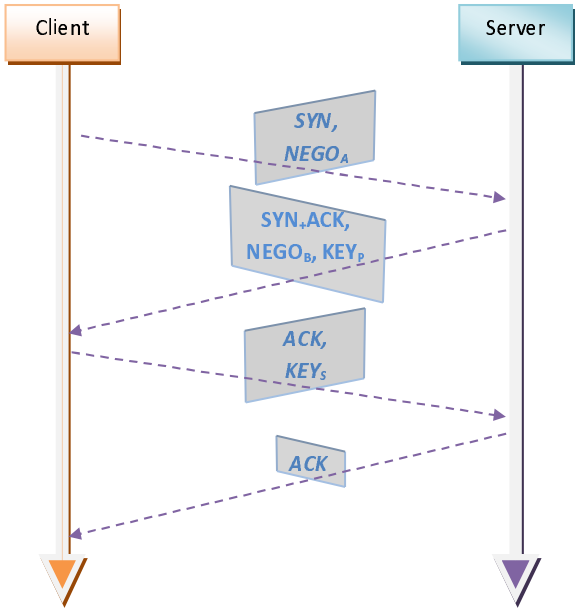

Figure 3.4: Secure TCP: negotiation data ($NEGO_x$) is initially exchanged, the remote endpoint sends its public key (KEY_p) so the local endpoint can afterward send an ACK segment with session keys (KEY_s), protected by the local private key and the remote public key. These session keys are used for confidentiality and integrity.

3.6 Proposed Internet Draft

A discontinued IETF Internet Draft [41] proposed an extension to support TCP port names. The main goal of this proposal was to increase the number of concurrent connections for existing services by decoupling them from fixed port numbers reserved by IANA.

In this proposal, named server ports are somehow *resolved* by clients, i.e., clients propose a resolution that is accepted by the server host, if the name exists. In fact, it is not exactly a resolution since the destination port is not taken into consideration. Thus, a SYN with a server named port contains also a proposed server port number, and the returned SYN+ACK returns a name-number acceptance reply. The modified *three-way handshake* is very similar to the original one, as shown in Fig. 3.5.

Names are defined within a *TCP portname* option as described in Table 3.1.

Kind (1 byte)	Length (1 byte)	Option Data (*portname's length* bytes)
(to be defined)	≥2	*UTF-8 portname*

Table 3.1: Definition (*kind, length and option data*) of the *TCP portname* option suggested by Dr. Joseph Touch. "Portnames" are limited to UTF-8 strings.

Clients would specify a port name, in the form ⟨*portname.portnumber*⟩, sent with the SYN segment header. One of these fields could be unspecified but the stack would substitute it either by the associated service name for the given *portnumber*, or by the port number for the given service name. Case no IANA "assigned" service name exists for the *portnumber*,

"UNKNOWN" would be chosen for *portname*. A similar concept applies when a *portname* is addressed without any specific *portnumber*, therefore a special predefined port number must be used to signal "whatever port".

The suggested behavior for servers was that they would bind to a specific port, like "0". This was just a way of informing the TCP stack that incoming connection requests, for the port name to which the server was bound, should be accepted independently of the destination port number.

Both source and destination TCP port numbers are randomly generated, meaning that the destination TCP port number can collide with some service running at the remote endpoint, if we have present that the server host might not be port name aware. More, in this case a standard SYN+ACK is returned because the server ignores the unknown *TCP portname* option; the client will notice the lack of this option in SYN+ACK and only then the connection will be aborted by sending a RST to the server. As shown, connections can be half-opened at the server host due to this, configuring a minimum DoS scenario completely under the control of the originator - he was the first sending the SYN segment with the *TCP portname* option and he is ultimately the last one responsible for aborting the connection.

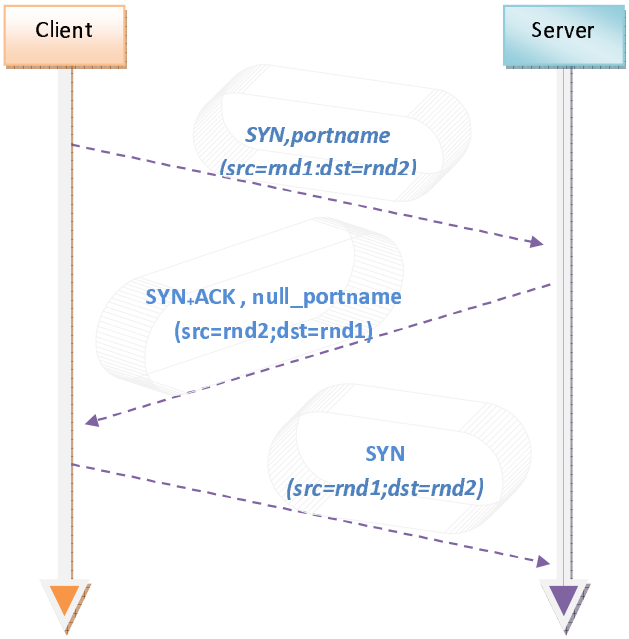

Figure 3.5: Joe Touch's proposed IETF draft *(draft-touch-tcp-portnames-00.txt)* for TCP port names. The name goes as a TCP option in the SYN and SYN+ACK acknowledges it, using the same type of option but with a zero-length portname option. Source (src) and destination (dst) port numbers are chosen randomly at start.

Port names are UTF-8 strings, not arbitrary data, corresponding to the keywords used to describe services in IANA's list of Reserved Ports, which can be rather restricted. They are exchanged within a TCP header option, which strongly limits their maximum length. As TCP headers are limited to a maximum of 60 bytes, 20 of them mandatory, TCP options can only occupy 40 bytes. Moreover, part of these 40 bytes may be occupied by other TCP options, which further reduces the possible lengths of port names.

Chapter 4

Architecture

4.1 Simple TCP Name Resolution

As a first approach, a simple TCP name resolution model (SRM)[13] was devised. It is effectively a name service for TCP ports that enables clients and servers to resolve arbitrary names (byte arrays) to TCP ports, using the following semantic: strict byte equality between names.

The advantages of using this name service are twofold: (i) users may discriminate servers using names instead of numbers and (ii) TCP port scanners, such as `nmap`, should not be capable of discovering servers bound to unusual names.

Using names for referring ports provides a more intuitive way to refer TCP services, instead of numbers. Service names that formerly were bound to static, well-known port numbers may continue to exist but do not need any more to be bound to the same ports. For instance, we can bind the names `http` to port 8080 and `http1` to port 80. Clients access either port specifying their name, `http` or `http1`, instead of numbers 8080 and 80. Port names are also useful for uniform and uniquely tagging ports used by the servers of overlay networks.

Using arbitrary byte arrays to name TCP ports also prevents port scanning tools to discover listening ports. In fact, the success of port scanners in discovering listening ports bound to servers is due to the current small domain of port numbers — $[1, 2^{16} - 1]$. With arbitrary port names, we are able to deploy services with unusual, possibly long port names which cannot be easily found by port scanners. Services with unusual, confidential port names may be useful in many circumstances requiring restricted access profiles, namely:

- Experimental server deployment in pre-production environments;
- Private service deployment, such as personal content providers (file or web servers, mail servers);
- Restricted overlay networks.

The screening of listening TCP ports by mapping them to unusual port names is somewhat similar to the Port Knocking mechanisms [3, 20]. However, Port Knocking is an access control mechanism that requires a per-host or per-network access key. Instead, in this model it is only required the knowledge of port names and not any key-based access control mechanisms; the knowledge of a port name is the key, though *weak*, to access the service that uses the port.

4.1.1 Name binding

This name service allows TCP servers to bind names to listening ports and clients to use port names when requesting a TCP connection. The port name is associated with the service/application during the socket binding procedure at the server side. Clients refer the port name when they specify the TCP address of the server in a socket connection request.

As the main focus in this model is to provide a TCP name service, and not to fully replace port numbers by port names, port names must always be associated to port numbers. Therefore, the modified TCP layer on the server side will keep a port number to each local port name. When a application binds a name to a TCP endpoint (socket), it immediately gets a number as well. For backward compatibility, applications may specify the port number; if not specified, the TCP layer allocates a free port number.

For applications binding only names to ports, and not fixed numbers, the TCP layer can also allocate random port numbers on a per-request basis. The benefits of this protocol decoupling from fixed port numbers are (i) harder traffic eavesdropping and (ii) increased number of concurrent connections, as in [41].

Port names are resolved as soon as possible to allow clients and servers to use port numbers in the normal TCP segment exchanges. Consequently, the name resolution is integrated in the TCP synchronization phase, where clients and servers exchange initial sequence numbers and requested/allowed TCP options (see Fig. 4.1). The port name is specified in the SYN request, the name→number resolution is given in the subsequent SYN+ACK segment. Thereafter, both peers will always use the server port number in all segments exchanged in the TCP stream. Stateful firewalls should have no problem managing this, if they're updated to keep track of port number/name pairs.

The name resolution works as follows. The server host, when receiving a SYN with a port name, ignores the destination port number (zero in Fig. 4.1) and looks for a socket in the LISTEN state bound to that port name. If such a socket is found, a SYN+ACK is sent to the client containing the name resolution, i.e. port name and number. Otherwise, a RST packet is sent with the port name, without resolving it.

4.1.2 Backward compatibility

Port name resolutions, expressed in SYN requests, should carry a null server port number to force a RST reply from standard TCP stacks. Though the TCP standard does not refer that TCP ports cannot use the number 0, it is not used in practice[1]; so it can be used as a mechanism to differentiate old TCP stacks from new ones implementing port names. Port scrubbers [22] should be updated to include TCP destination port 0 as valid when the port name option is present.

Port name resolutions are only provided to clients that request a connection to a TCP endpoint with a port name. Clients using the normal TCP connection request, i.e. using a server port number, do not get any name resolution when the port number is actually associated with a name. Likewise, they do not receive a port name in a RST reply. This is done for two reasons. First, the client did not request a name resolution, so it should not get one. Second, for backward compatibility with current TCP stacks, clients using the

[1]IANA currently reserves the use of port 0 mainly for historic reasons, though it can be used to signal a future expansion of the port numbering space.

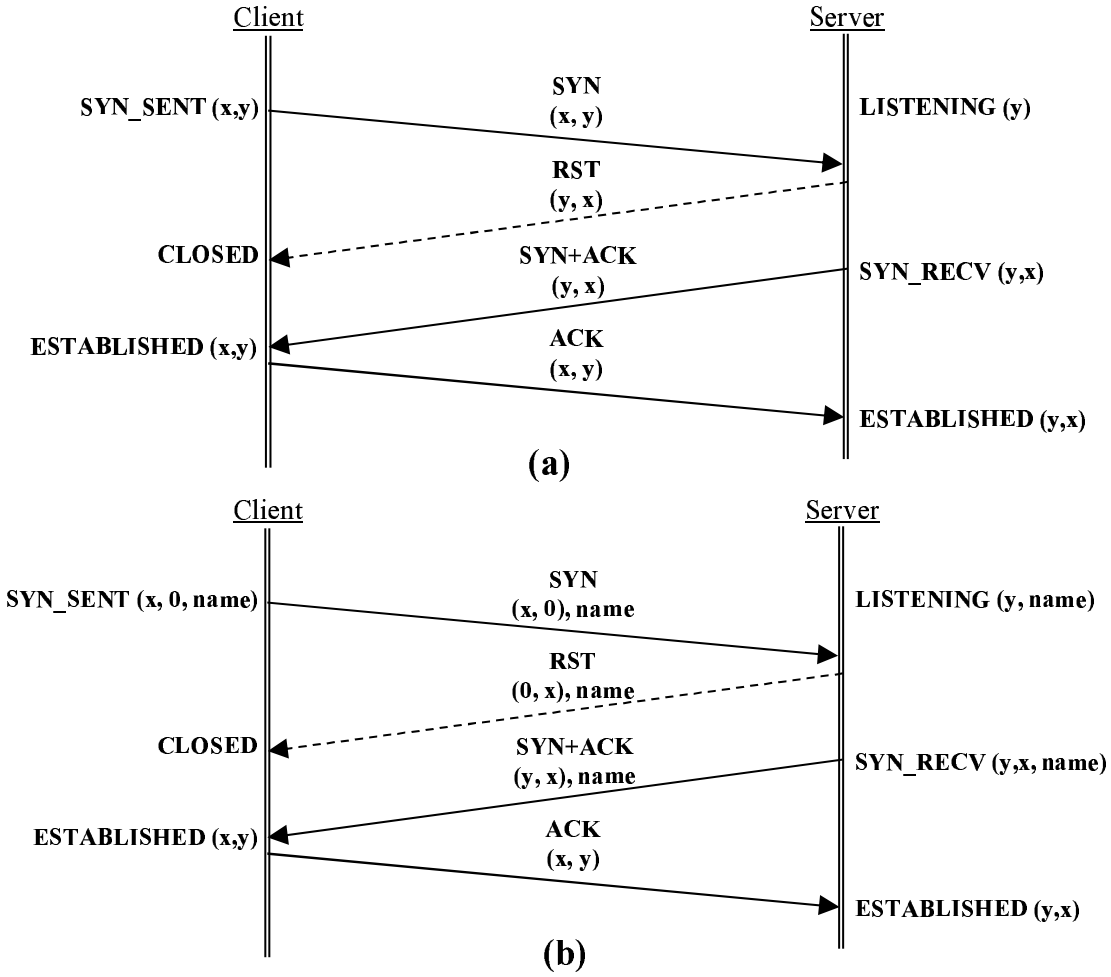

Figure 4.1: Standard *three-way handshake*, using port numbers x and y (a) and extended *three-way handshake* using a server port name (b). The slashed line represents an alternative server reply (RST segment) when the connection to port y or with the given name is not possible or allowed.

actual number-based port addressing should observe the standard TCP behavior from servers (described in Fig. 4.1).

4.1.3 Port names in TCP segments

As above described, port names are only used in the TCP synchronization phase, namely in SYN, SYN+ACK and RST segments. Thus, we have to conceive a way of adding arbitrary names to these segments and to signal that they should be used.

As referred in section 3.6, adding names to TCP headers is not a suitable solution, because it severely limits the length of port names. Instead, we decided to use the TCP payload to store the port name and to use a new TCP option to signal the presence and the length of the port name in the beginning of the segment payload.

TCP synchronization segments allow clients and servers to exchange data. Though this is not supported by the Berkeley sockets API, it is allowed by the XTI (X/Open Transport

	TCP standard		TCP with port names	
Segment	seq	ack	seq	ack
SYN	ISN_c	0	ISN_c	0
RST	0	ISN_c+1	0	ISN_c+1+ len(name)
SYN+ACK	ISN_s	ISN_c+1	ISN_s	ISN_c+1+ len(name)
ACK	ISN_c+1	ISN_s+1	ISN_c+1+ len(name)	ISN_s+1+ len(name)

Table 4.1: Calculation of sequencing numbers (*ack* and *seq*) for the *Simple TCP Name Resolution* model.

Interface) and is a correct behavior according to the standard [28]. RST segments usually do not carry any data in their payload but a standard amendment [4, §4. 2.2.12] defines that they may carry a reason message in their payload. Thus, using the payload of synchronization and reset segments for exchanging port names is correct, according to the standards, though such data should not be delivered to upper layers.

As port names are transmitted in the segments' payload, they have a direct effect in the management of the stream sequence numbers. The sequence and acknowledgement numbers exist to control the data stream between peers, guaranteeing byte order and resilience to data loss.

It was decided to keep the semantics of sequence numbers during the synchronization phase using port names, i.e. names are seen as ordinary data exchanged in payloads (see Table 4.1), but they are removed from the data that is provided to upper protocol layers. This does not raise any coherence problem, since upper layers are not aware of sequence numbers. In other words, upper layers do not notice that the amount of data received in segments' payload is not the same amount of data they actually get.

Consequently, whenever a name resolution is required in a SYN segment, its reply, either a SYN+ACK or a RST segment, acknowledges a sequence number equal to the client's ISN plus the port name length plus one. Similarly, the client's ACK will contain a sequence number equal to its ISN plus the server's port name plus one. In this case, the acknowledgement number equals the server's ISN plus the port name length plus one.

4.1.4 Managing port access restrictions

The TCP name service prevents services with unusual names to be discovered by port scanning tools, hiding them from people or tools that wish to exploit and not really use them. But in 4.1.1 we saw that port names may be associated to fixed numbers bound to listening ports. Thus, to enforce the name-based access control we need to disallow clients to connect server ports using only their number.

Consequently, servers must specify, when binding, a name to a listening port, if (i) number-based connections are permitted or (ii) only name-based connections are allowed. In this last case, the TCP layer is free to use one random number for the server port per name-based connection request.

4.1.5 Caching of name resolutions

Some name services' clients maintain caches of name resolutions. For instance, this is common in DNS but not in RPC. It was decided not to maintain caches in the TCP port name resolutions.

Two main reasons justify this decision. First, name resolutions are piggybacked in the two first segments of a TCP synchronization, thus the overhead of name resolution is too reduced to justify the existence and management of caches in clients for increasing performance. Second, stalled cached resolutions can lead to wrong TCP connections that only applications can possibly detect, but not the TCP layer itself.

4.1.6 Windowing adjustments

One adjustment needs to be made in the TCP windowing mechanism, so it can deal with segments that transport data during the *three-way handshake*. The change concerns the *receive window* size. The client defines and advertises this value when starting a connection while the server's *receive window* size is announced in the SYN+ACK segment. In this specific model, only SYN and SYN+ACK segments contain the port name payload. Therefore, in order to process the named SYN+ACK the client needs to adjust its *receive window* to an appropriate and suitable value (at least the port name length plus TCP header). The server can apply a similar value to its *receive window*.

There are some implementation constraints and drawbacks regarding the choice of MSS as discussed latter on Chapters 5 and 6, respectively.

4.2 Enhanced TCP Name Resolution

SRM may be an interesting model due to its simplicity but that turns out to be its major disadvantage. The Enhanced TCP Name Resolution (ERM) is an improvement of the previous model. It works out being a generalization of SRM by removing the strict concept of being just a mere resolution mechanism. The meaning of port names was also extended to include semantics. In fact, the concept of a "port name" is overridden when transporting data that can be used for any purpose (e.g. resolution, validation, authentication). Note, however, that the key points of SRM are also present or can be addressed by using ERM, specifically the possibility of using friendly names to tag services and the existence of implicit security by enforced obscurity (i.e. services are not visible if not properly addressed).

The advantages of using this enhanced model over SRM are:

- totaly decouples the TCP connection establishment from strict TCP port addressing;
- supports other features besides name resolution;
- delegates the connection establishment logic to external processes;
- supports a pluggable logic mechanism closely integrated with the TCP/IP stack.

Standard TCP connections succeed if packets are routed to endpoints with listening TCP services. With SRM the connection establishment depends on an in-band name resolution, translating names into port numbers, whereas in ERM connections are established if the logic implemented in external user processes decides that the *three-way handshake* should proceed normally. Ultimately, this scheme resolves a connection request to a listening TCP service and uses exchanged data ("port names") to validate the connection establishment.

A simple name resolution as the one used by SRM does not provide the security that is demanded for establishing secure and trusted communications, namely through authentication or authorization techniques; but all this can be achieved using ERM. In fact, port names exchanged during the *three-way handshake* may contain encrypted data.

As already mentioned, it is up to specific external processes to decide if the connection goes through or not. These processes implement some logic based on port names. A Domain of Interpretation (DOI) corresponds to the definition of a set of rules which allow service addressing/resolution during the *three-way handshake* and that may conditionate a successful connection establishment. Two DOI examples are the exact name match resolution logic or a name resolution with authentication based in a pre-shared key. The processes responsible for implementing DOI rules are called DOI Resolvers.

For a successful communication between endpoints, a DOI must have the same meaning at each endpoint. Therefore, the DOI is identified by a number that must be managed centrally by some entity, like IANA. Communication attempts between endpoints that do not understand some DOI would fail.

4.2.1 Name binding and connection establishment

Being a generic mechanism using names to address and validate connections between TCP endpoints, it makes even more sense to integrate it directly into the TCP synchronization phase, as presented in the extended *three-way handshake* of Fig. 4.2.

Similarly to SRM, this name service allows TCP servers to bind names to listening endpoints during the socket binding procedure. Clients do not need to use the same port name as the one used by the server for name binding, since there is an effective name resolution in the server. Instead, clients send some port name payload that will be used by the server's DOI Resolver to find out, accordingly with that DOI's logic, the listening endpoint at the server side. So, the first step here involved is effectively a name resolution. Afterwards, data built as a port name structure is exchanged in the remaining SYN+ACK and ACK segments. That data is used to validate the connection to the endpoint that was found in the first step, after a named SYN request.

The name resolution and connection validation procedures are processed by DOI Resolvers. These use the information contained within port names to decide if the connection proceeds or not. For each connection attempt there is not just one decision, there are in fact three, one after each segment of the *three-way handshake*. Due to this, besides taking a decision, a DOI Resolver may need to produce an output that will be transmitted as a port name in the subsequent segment of the *three-way handshake* and that will serve as input to the next decision. One consequence of this interaction between the TCP stack and DOI Resolvers during the connection establishment phase is that processing of incoming packets must be asynchronous. During the *three-way handshake* a segment is handled as usually, typically by the TCP layer, but since it may interact with a DOI Resolver, its processing has to be interrupted and suspended until getting an answer from the DOI Resolver. This can impose some implementation problems, depending on the architecture of the underlying kernel and TCP stack.

Here port names are always present in the three segments, or in RST, even if their "port name payload" is empty (i.e. only composed by a port name header). The reason lies in the generic architecture of this model, whose resolution/validation logic is implemented by DOI Resolvers. Thus, the TCP stack is not aware of that logic and, as such, it does not know if all

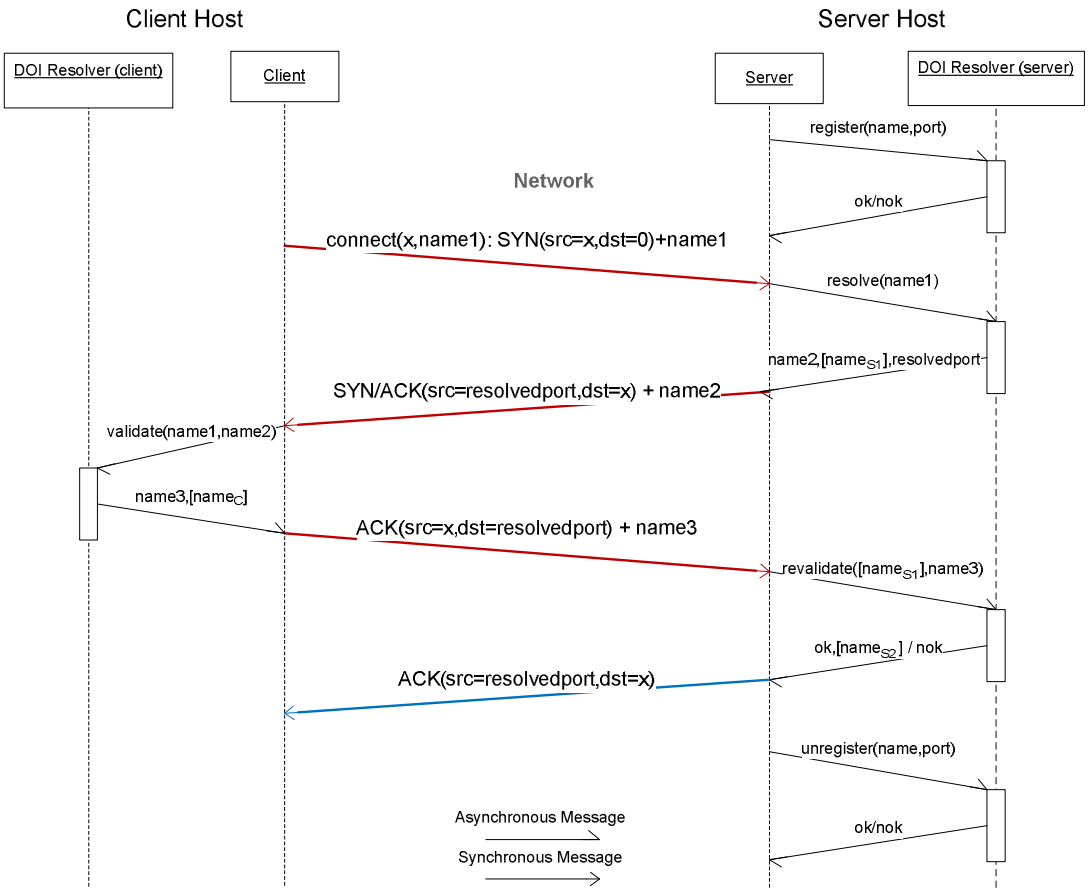

Figure 4.2: Extended *three-way handshake* for the Enhanced Resolution Model and its integration with DOI Resolvers. Each segment has a source and destination TCP port number *src/dst* and may have some payload. Communication with DOI Resolvers is made through *resolve()*, *validate()* and *revalidate()* primitives.

or part of the exchanged port names are used to complete and accept the connection. This is also the reason why port access restrictions, as the ones depicted in section 5.1.4, do not make sense in this scenario: servers cannot be addressed by port number.

Port names are actually structures formed by a port name header, used for "addressing" purposes, and some attached data, as shown in Fig. 4.3. As port numbers are not used to correlate SYN and SYN+ACK segments, neither their payload through a strict comparison, as happens with SRM, a port name must contain an identifier that must be preserved across those segments. In fact, the client chooses an identifier per connection and thereafter that identifier is used on subsequent named segments. The structure of port names must also contain a field telling which DOI the port name is related to, so it can be delivered to the proper DOI Resolver. These two fields, `identifier` and `domain of interpretation`, form the port name header which will be used to correlate named SYN and SYN+ACK segments instead of just using the mentioned `identifier`.

Three special names can be observed in Fig. 4.2: $name_{s_1}$, $name_{s_2}$ and $name_c$. The first two correspond to a name given to the server endpoint, in the server context, by the local DOI Resolver and the latter is assigned by the client's DOI Resolver to the server endpoint. Both are

Figure 4.3: Structure of port names in the enhanced model context.

optional and therefore not essential in this model; they can be obtained by *getsockname()* and *getpeername()* glibc functions for specific application processing after connection establishment but are not used to conditionate its own success. The client port name ($name_c$) can be assigned right after receiving the named SYN+ACK. At the server side it is not so clear when to assign a port name to its server endpoint. There are two options: (i) after receiving a successful SYN, with $name_{s_1}$, or (ii) after successfully receive the subsequent ACK, with $name_{s_2}$. This decision is taken by the DOI Resolver.

The connection establishment involves DOI Resolvers and works as follows. Initially there must be a DOI Resolver, responsible for a certain domain of interpretation, running locally to both endpoints. When the TCP server executes the socket binding procedure, it registers a name and, eventually a port number. Consequently, the server registers itself in the DOI through the *register()* primitive, using the given port name and the the listening TCP port number as arguments.

A named SYN, with the port name payload referred as *name1* and with destination port number equal to 0, is sent to the server host. This one will invoke the *resolve()* primitive in the DOI Resolver, returning *name2* to be sent with the SYN+ACK and a resolved port number corresponding to that service. It may also return $name_{s_1}$ to be locally assigned to the server endpoint or leave that decision to a latter time, when dealing with the client's named ACK. The client, upon receiving the named SYN+ACK, will invoke the *validate()* primitive in its own resolver. This resolver understands the port name because it handles requests for the same domain of interpretation as the server's DOI Resolver does. The validation arguments include *name1*, sent with the named SYN, and *name2* that is returned in the SYN+ACK. The client's DOI Resolver will then return *name3* to be sent with the ACK segment and may return $name_c$ to be assigned to the server's endpoint, in the client's context. Finally, the server receives the named ACK and makes a final validation through the *revalidate()* primitive on the DOI Resolver. The latter will decide wether the connection establishment should or not succeed and may also return $name_{s_2}$ to be assigned to the server endpoint instead of $name_{s_1}$. This leads to a final standard ACK sent towards the client or eventually a standard RST if it was decided to abort the connection. The last ACK segment is a natural consequence of acknowledging data from the previously received named ACK.

When the listening endpoint is terminated, the TCP layer will locally deregister its name from the DOI Resolver by invoking the *unregister()* primitive.

4.2.2 Backward compatibility

Legacy hosts must observe the traditional TCP stack behavior. The same principle that was used for the simple model also applies here: a named RST is sent only for unsuccessful named SYN connection requests (e.g. due to failed name resolution). The port name used in the RST does not need any port name data, although it can be used to explain the reset cause as RFC 1122 [4] suggests.

While SRM allowed legacy connections to be established to named servers, ERM does not allow it. Therefore, a non-named connection attempt will lead to a standard RST segment.

	TCP standard		TCP with port names	
Segment	seq	ack	seq	ack
SYN	ISN_c	0	ISN_c	0
RST	0	ISN_c+1	0	ISN_c+1+ len(name1)
SYN+ACK	ISN_s	ISN_c+1	ISN_s	ISN_c+1+ len(name1)
ACK	ISN_c+1	ISN_s+1	ISN_c+1+ len(name1)	ISN_s+1+ len(name2)
2^{nd} ACK	N/A	N/A	ISN_s+1+ len(name2)	ISN_c+1+ len(name1) + len(name3)

Table 4.2: Calculation of sequencing numbers (*ack* and *seq*) for the *Enhanced TCP Name Resolution* model.

4.2.3 Port names in TCP segments

Similarly to SRM, in ERM port names are exchanged during the TCP synchronization phase. Unlike SRM, port names are present in every segment that may be exchanged during the *three-way handshake*: SYN, SYN+ACK, ACK and RST segments.

In the current model, every segment has a port name payload, even in the ACK segment. As mentioned earlier, this data must not be seen by upper protocol layers and should be removed from receive buffers. Additional data following the port name, should be handled normally (e.g. a named ACK with some other payload should be treated as a standard ACK segment with that same payload). In other words, port names should be discarded from normal data processing.

As happens with SRM, sequence numbers must be correctly updated to handle the transmission of data in the segment's payload. As shown in Table 4.2, whenever a named connection request is issued through a SYN segment, its reply, either a SYN+ACK or a RST segment, acknowledges a sequence number equal to the client's ISN plus the "name1" port name length plus one, $ISN_c + 1 + len(name1)$. This also corresponds to the value used for the sequence number of the client's ACK that follows. Its acknowledgement number is equal to $ISN_c + 1 + len(name1)$.

Standard *three-way handshakes* just use three segments, although if the last ACK has some payload then the server will consequently acknowledge it, using another ACK. Thus, the second ACK occurs naturally, it is not required by the name resolution protocol. Its sequence number equals $ISN_s + 1 + len(name2)$. This ACK segment acknowledges "name3" sent with the first ACK, therefore its acknowledgement number is $ISN_c + 1 + len(name1) + len(name3)$.

4.2.4 Caching of name resolutions

It is worth remembering that during the *three-way handshake* segments are correlated using an identifier inside the port name header. As mentioned earlier, there is no explicit name resolution. DOI Resolvers can implement more or less complex logic, depending on implementation. As such, caching does not make any sense in this scenario.

4.2.5 Windowing adjustments

Some adjustments must be taken in TCP windows for proper functioning. All *three-way handshake* related segments, as told before, have port names which may be considerably large. Therefore, not only the client's *receive window* must be big enough but also the server's *receive window*. It was chosen the maximum safe value allowed for the TCP window, as will be detailed in Chapter 5. The use of the Window Scale option was discarded (i.e. considered zero) since endpoints may not support it. That way, it was followed the standard approach of setting a high value for the advertised TCP window. Actually, this will be more than enough as discussed in Chapter 6. Worth noting is that strictly enforcing a high window may go against some RFC recommendations [1] and can interfere with congestion avoidance algorithms.

4.2.6 DOI enabled Resolution Models

The DOI paradigm can be used to provide resolution models with more or less complex logic for addressing and validation. Some different use cases are here briefly presented.

Addressing by exact name

This use case corresponds to the typical name resolution scenario used by standard name resolution protocols/systems, where a name is resolved to some sort of address. Transposing this to TCP means that services can be addressed by a name specified in the connection request. The resolution consists on finding a TCP endpoint in the server whose assigned name equals the asked name. SRM does exactly this, but it was built into the TCP stack. Conceiving a DOI version for this use case is quite simple. It would consist of:

1. defining the DOI logic to:
   (a) upon *register*, record the name used by services during the socket binding procedure;
   (b) lookup a service with the same port name data that comes with the named SYN;
   (c) if such a service is found then allow the connection to proceed when returning from *resolve*;
   (d) to acknowledge positively both named SYN+ACK and ACK segments, during *validate* and *revalidate* primitives respectively;
2. sending a named SYN with the name of the pretended service.

TCP segment	Port Name Data
SYN	*name*
SYN+ACK	N/A
ACK	N/A

Table 4.3: Content of port name data in the *addressing by exact name* scenario.

Addressing by regular expression

In this scenario a service is addressed using a regular expression (*regex*). The model is similar to the previous one just differing on how the service is matched against the given name.

Consider a set of services providing the same functionality, that are bound to the names `http1`, ..., `httpN`. A way of providing a very basic form of load-balancing would be to issue a connection request using the *regex* "http.*". A generalization would be a friendly service addressing (e.g. users wishing to connect to some service would specify some characteristic about that service, translated in a *regex*).

TCP segment	Port Name Data
SYN	*regex*
SYN+ACK	*resolved name*
ACK	N/A

Table 4.4: Content of port name data in the *addressing by regular expression* scenario. The SYN port name data contains a regular expression used to find the correct service in the server host. The service resolve name is returned in the SYN+ACK.

Another implicit use case would be the addressing of services with multiple versions. If the name syntax is in the form of *servicename.version*, then users or entities could specify the name "xpto.*" to address any version of that service or could instead use "xptoV10" to address the version 10 of service with the assigned name "xpto".

The steps to achieve these goals would be:

1. defining the DOI logic to:
   (a) upon *register*, record the name used by services during the socket binding procedure;
   (b) lookup a service using the port name data that comes with the named SYN as a *regex*;
   (c) if such a service is found then allow the connection to proceed when returning from *resolve*, besides returning the resolved name;
   (d) to acknowledge positively both named SYN+ACK and ACK segments, during *validate* and *revalidate* primitives respectively;
2. sending a named SYN with a name matching the *regex* request.

Secure access

This use case ensures a secure access to services by enforcing a protected connection establishment where both endpoints are mutually authenticated. MAC values can be exchanged consecutively [43] to validate a dialogue, in this case the messages exchanged during the *three-way handshake*.

We assume that a shared key k was previously distributed to both endpoints by some out-of-band process. The shared key is never exchanged between endpoints during the connection establishment. It will be used by each endpoint to compute a MAC sent in the subsequent

segment and/or validate a MAC that came in a former segment. Random numbers are consecutively added to port names and serve as input for the current MAC calculation.

For the following explanation, *name1*, *name2* and *name3* refer just to the port name data, thus port name header is excluded hereafter. The port name data format depends on the segment exchanged. In a SYN segment it has the $\langle \text{name,R}_1\text{,MAC}_k(\text{name,R}_1)\rangle$ format, where R_1 is a random number and *name* is a service given friendly name. The SYN+ACK will introduce another random number, R_2, which together with *name1* will be used for posterior validation. Therefore, the port name data has the $\langle \text{R}_2\text{,MAC}_k(\text{R}_2\text{,}name1)\rangle$ format. Finally, the named ACK correlates with the previous segments by also using a random number, R_3, and use it as input for MAC calculation with the previously received port name data. This port name data follows a $\langle \text{R}_3\text{,MAC}_k(\text{R}_3\text{,}name2)\rangle$ format.

TCP segment	Port Name Data
SYN	$name1{=}name,R_1,MAC_k(name,R_1)$
SYN+ACK	$name2{=}R_2,MAC_k(R_2,name1)$
ACK	$name3{=}R_3,MAC_k(R_3,name2)$

Table 4.5: Content of port name data in the *secure access* scenario.

In terms of segment processing, the necessary steps here involved are:

1. the client computes a random number R_1 and uses the shared key k to compute a MAC of *name* and R_1; a named SYN is sent with the corresponding port name data;

2. the server validates the received port name data, by replaying the MAC calculation;

3. the server computes a random number R_2 and uses the shared key k to compute a MAC of R_2 and *name1*; a named SYN+ACK is sent with the corresponding port name data;

4. the client validates the received port name data, by replaying the MAC calculation;

5. the client computes a random number R_3 and uses the shared key k to compute a MAC of R_3 and *name2*; a named ACK is sent with the port name data formed by these values;

6. the server validates the received port name data, by replaying the calculation of the MAC and replies with a standard ACK if it succeeds;

All port names are calculated using the shared key. As the SYN's port name is specified directly by the TCP client application, this may raise a problem concerning where k effectively resides (DOI Resolvers vs. client/server applications).

Two solutions are possible to make DOI Resolvers aware of the shared key k:

1. have a specific DOI for each possible shared key;

2. make the client register a port name for itself containing k that would have to be passed in the *validate* primitive as another argument. Similarly, during the name binding procedure in the server, the bound name would have to contain k.

Key distribution

This use case uses the asymmetric Diffie-Hellman [9] key agreement protocol to generate a shared key to be used between both endpoints. The key can be obtained by applications for whatever purpose (e.g. traffic confidentiality).

The port name data format depends on the segment exchanged. In the SYN segment it corresponds to a friendly *name* that is going to be used to find the appropriate service during name resolution. The server will send a SYN+ACK with a port name data $Y_S = \alpha^s \bmod q$, where q is a sufficiently large prime number, α is a primitive element of *GF(q)* and s is a server generated random number such as $1 \leq s \leq q-1$. Upon receiving SYN+ACK, the client can obtain a shared session key $k = Y_S^c \bmod q$, where c is a client generated random number such as $1 \leq c \leq q-1$. Then the client sends an ACK whose port name data $Y_C = \alpha^c \bmod q$. Finally, the server obtains the shared key $k = Y_C^s \bmod q$.

TCP segment	Port Name Data
SYN	*name*
SYN+ACK	$Y_S = \alpha^s \bmod q$
ACK	$Y_C = \alpha^c \bmod q$

Table 4.6: Content of port name data in the *key distribution* scenario.

What is assumed is that both endpoints know α and q, which can be public without any security concern. The port name data (e.g. Y_C,Y_S) may have a length of 128 bytes, assuming a prime number q of 1024 bits.

The shared key k corresponds to $name_c$ and $name_{s_2}$ obtainable using *getpeername()* and *getsockname()* functions on client and server endpoints, respectively.

The described use case considers independent DOI Resolvers, i.e. distinct processes from client and server applications. However, there is no reason that limits the DOI Resolver to be built into the client/server applications themselves. Considering such a scenario, the model could be slightly modified. The Y_C value may be immediately calculated and sent in the SYN and thus the client may obtain the shared key k when receiving the SYN+ACK segment. Similarly, the server obtains k upon reception of the named SYN.

Considering a key distribution scenario where DOI Resolvers are built into applications, the content of port name data for the *three-way handshake* segments could be the one depicted in Table 4.7. In this case, the server obtains the shared key from $name_{s_1}$.

TCP segment	Port Name Data	
SYN	*name*,Y_C	$Y_C = \alpha^c \bmod q$
SYN+ACK	Y_S	$Y_S = \alpha^s \bmod q$
ACK	N/A	

Table 4.7: Content of port name data in the *key distribution* scenario with the DOI Resolver functionality implemented directly by the TCP client and server.

Chapter 5

Implementation

Our TCP name service was implemented in one of the latest versions of the Linux kernel (2.6.22.9), using a Fedora 7 kernel source package (kernel-2.6.22.9-91.fc7.src.rpm) as basis. In this chapter we will briefly describe the major kernel modifications, introduced in the TCP stack, to support the proposed name service.

In Linux there are two separate TCP implementations, one for IPv4 and other for IPv6. Since our implementation is mainly a proof of concept, we only updated the TCP version for IPv4. Nevertheless, we took into consideration some IPv6 issues, as for the naming of TCP endpoints, described in section 5.1.5.

5.1 Simple TCP Name Resolution

5.1.1 Socket related structures

Port names, either bound to local ports or to be resolved within TCP connection handshakes, are stored in memory areas dynamically allocated in kernel space. The structures that store port numbers, `inet_sock` and `tcp_sock`, were updated to include an optional port name, i.e. a pointer to one of those memory areas and the length of the name; `tcp_sock` includes also port access restrictions.

Some auxiliary structures, `tcp_request_sock` and `tcp_options_received`, were also enriched to maintain the length of the port name. This value simplifies the calculation of sequence numbers in TCP segments containing port names and helps locating the socket structure in hashed lists (see section 5.1.3).

5.1.2 Name→number mappings

Since port names are just a step towards port numbers, some mapping table converting names to numbers must exist. This mapping only applies for local ports, as no caching of remote name-number bindings is required. We implemented this mapping by extending the `inet_hashinfo` structure to include an hashed variable based on a new structure named `portname_hashbucket` as follows.

```
struct portname_hashbucket {
  spinlock_t         lock;
  struct list_head   list;
  unsigned short     port;
  char               * portname;
  unsigned short int portname_len;
};
```

An element is first added to this hashed variable when a socket is bound to a port name, in *inet_bind*. This is implemented by a new function, *inet_bind_hash_portname*, which extends the functionality of *inet_bind_hash*. An element is removed from the hashed variable when the socket referring it is eliminated in *inet_unhash*.

This hash variable is used for name lookup in two different occasions: (i) when binding a name to a port, in *inet_csk_get_port*, to check if the name is not already being used, and (ii) in *__inet_lookup_listener*, to obtain the port number of the listening socket upon receiving a SYN segment with a port name.

5.1.3 Socket hashed lists

Linux implements several hashed lists to index sockets. One of them is *listening_hash*, containing `INET_LHTABLE_SIZE` lists of sockets involved in TCP connections. The actual list of a socket is given by the functions *inet_ehashfn* and *inet_sk_ehashfn*, which produce an integer from the source/destination addresses and port numbers. Thus, both local and remote port numbers are crucial for indexing sockets in *listening_hash*.

However, clients using name-based connections raise a problem: they do not have a remote port number when adding a socket in the SYN_SENT state, i.e. after sending a SYN segment with a port name. Thus, for these client sockets there is a temporary hashing within *listening_hash* until getting the name resolution. This temporary hashing applies only to sockets in the SYN_SENT state; after getting a correct SYN+ACK segment with the required name resolution, the remote port is used to relocate the socket, now in the ESTABLISHED state.

For the temporary hashing we used the same functions and replaced the server port number by the port name length. We could as well have used a null server port number but using the name length is more likely to improve, with no extra costs, the spreading of sockets (only in the SYN_SENT state) among the hashed lists when many port names are used.

5.1.4 Defining port access restrictions

Implementing TCP port access restrictions is accomplished by setting a new, TCP level socket option. We named this option `TCP_BIND_PORTNAME` and gave it the value 15, which is unallocated. There are three different listening modes that can be set through this option, though the first one does not apply to port name binding:

- 0 → port number only (current standard);
- 1 → port name only: legacy connection requests to the related port number are not allowed;
- 2 → port number and port name: both legacy and name-based SYN requests are accepted.

This socket option is checked in the *tcp_v4_rcv* function, upon the arrival of a SYN segment.

Kind (1 byte)	Length (1 byte)	Option Data (2 bytes)
0x45	4	···

Table 5.1: Definition (*kind, length and option data*) of the "Port Name" TCP option.

5.1.5 Using TCP port names

For binding names to TCP ports and to express port names when connecting to them, we created two new structures by extending `sockaddr_in` and `sockaddr_in6` structures, for IPv4 and IPv6, respectively. The new types were created by adding two extra fields: a pointer to the port name and the port name length, as shown on the data structures here presented.

```
struct sockaddr_in_named {
  sa_family_t    sin_family;       /* AF_INET_NAMED     */
  in_port_t      sin_port;         /* port number       */
  struct in_addr sin_addr;         /* IP address        */
  unsigned char  sin_zero[2];
  uint16_t       sin_portname_len; /* Port name length  */
  char __user    *sin_portname; /* Pointer to port name */
};

struct sockaddr_in6_named {
  sa_family_t     sin6_family;       /* AF_INET6          */
  in_port_t       sin6_port;         /* port number       */
  uint32_t        sin6_flowinfo;     /* flow information  */
  struct in6_addr sin6_addr;         /* IPv6 address      */
  uint32_t        sin6_scope_id;     /* Scope ID          */
  uint16_t        sin6_portname_len; /* Port name length  */
  char __user     *sin6_portname; /* Pointer to port name */
};
```

Furthermore, two new address families were created for using with these new structures: `AF_INET_NAMED` for IPv4 and `AF_INET6_NAMED` for IPv6. Note that at kernel level these new address families are used solely for identifying the type of naming structures provided by client applications. For all other family tagging requirements, it continues to use the constants `AF_INET` and `AF_INET6`.

The function *inet_bind* was extended to support binding to a local port name. Similarly, the functions *tcp_connect* and *tcp_v4_connect* were likewise extended to support the connection to named ports.

5.1.6 Port names in TCP segments

When a client application issues a connection request using a port name, the local TCP stack copies the port name from the `sin_portname` field of the provided `sockaddr_in_named` structure and updates internal variables as needed. The kernel will then send a SYN packet with the port name in the payload and a TCP option indicating the length of the port name, which corresponds to the given `sin_portname_len`.

We used the value `0x45` to define a new TCP option, `TCPOPT_PORTNAME`, which uses a 16-bit integer to describe a port name length.

As the Linux TCP stack does not handle the exchange of user data in synchronization segments, no modifications were required to prevent port names exchanged in SYN and SYN+ACK segments to be delivered as normal data to applications. Only port names in RST segments had to be removed from the data provided as reset reason.

Linux has one special socket *tcp_socket* that is only used for sending RST segments by the function *tcp_v4_send_reset*. This function and another one, *ip_send_reply*, that it calls to generate the actual RST IP datagram, were extended to process further parameters, namely port names, and to handle port names in TCP segments.

5.1.7 IP fragmentation

The Linux TCP implementation sets the Don't Fragment IP flag in TCP segments not containing payload, such as SYN and SYN+ACK, which is a correct behavior [27] since their packet length is bellow the fragmentation threshold of 68 bytes. But with our port name resolution their payload contains a (possibly large) port name, thus fragmentation must be allowed in those cases. The function *tcp_transmit_skb* was modified, when calling *ip_queue_xmit*, to allow IP fragmentation for these synchronization segments whenever their size surpasses the fragmentation threshold.

5.1.8 Summary of source code changes

The main changes and enhancements were the following:

- create and use an hash to map port names into port numbers;
- extend internal structures and functions to support port name data;
- change some function definitions to pass and deal with more arguments;
- create a few auxiliary functions to more easily deal with port names.

A detailed description of changes made within each kernel file can be found in Appendix A.

5.2 Enhanced TCP Name Resolution

5.2.1 DOI integration

In this model the TCP stack interacts closely with DOI Resolvers for assistance on name resolution. While on Linux the first runs on kernel (memory) space, DOI Resolvers run on user space as user processes. A DOI interface kernel module, described later in section 5.2.6, acts as a mediator between those two distinct areas, as shown in Fig. 5.1.

The kernel↔DOI communication occurs by means of a protocol based on Netlink sockets, properly detailed in section 5.2.8. The advantages of using a kernel module (LKM) are inherent to the modular concept itself and to the built-in logic. Therefore, a LKM can provide:

- a pluggable mechanism by means of dynamic (un)loading;
- a fault-tolerant architecture capable of handling problems with DOI Resolvers, such as crashes;
- an efficient, reliable and buffered asynchronous protocol using Netlink sockets.

A successful enhanced *three-way handshake* using this approach is seen in Fig. 5.2 along with the entities that take part in the process of connection establishment.

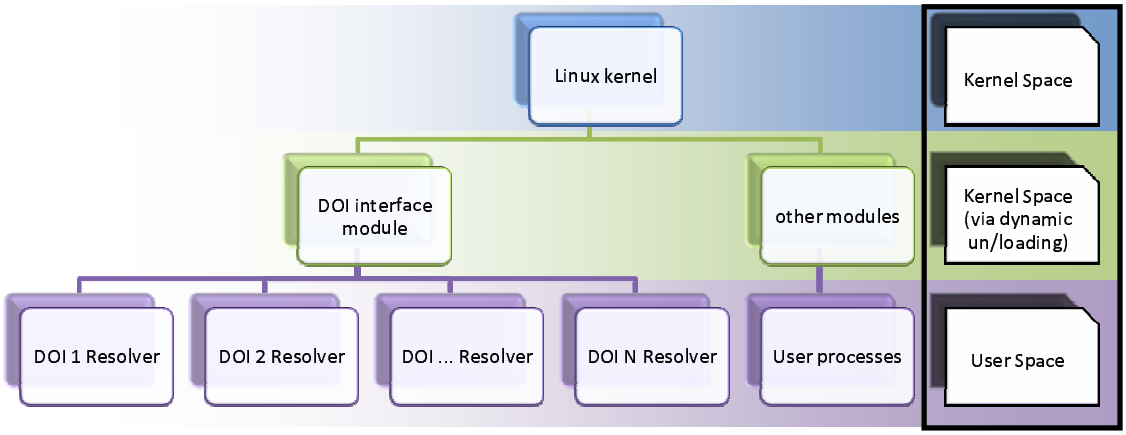

Figure 5.1: Layered hierarchy (kernel↪DOI interface kernel module↪DOI resolvers) involved in the interaction between the Linux kernel and DOI resolvers.

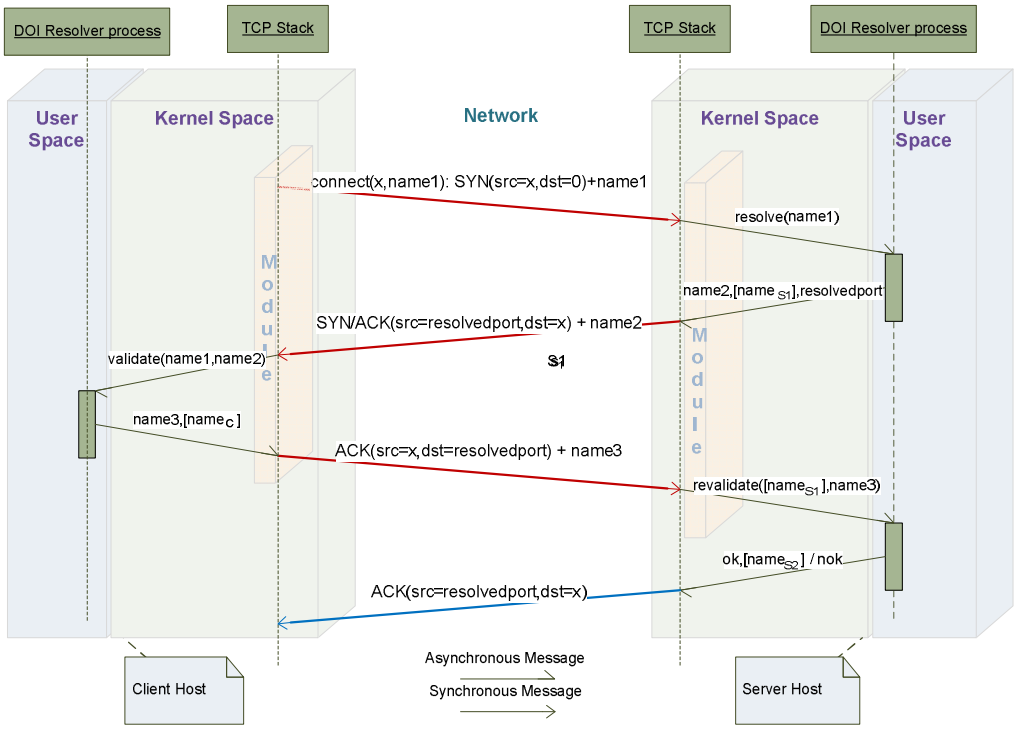

Figure 5.2: An implementation of ERM with the *three-way handshake* segments interacting with DOI Resolvers through a DOI interface kernel module.

5.2.2 Socket structures and port names

The structure that stores information of INET sockets,`inet_sock`, was updated to include more port name related variables, namely *resolved_portname, acked_portname, client_portname, server_portname* which correspond to the *name2, name3,* name_c and $name_{s_1}/name_{s_2}$ on Fig. 4.2, respectively. Their lengths were also included. These port names are intended to store temporary information used during packet processing. The port access restrictions

defined in `tcp_sock`, though meaningless in this model as mentioned before, were kept.

The structure `request_sock_ops` had to be slightly modified for adding support for port name parameters in the *send_reset* declaration. The reason for this lies in the need for sending RST segments using the *request socket*[1] in the asynchronous packet handling.

The purpose of the names assigned to client/server endpoints, $name_c$ and $name_{s_1}/name_{s_2}$ respectively, may concern some application logic derived from the output of DOI Resolvers; they do not influence the decision logic of the connection establishment. These names can be obtained by using the *getsockname()* and *getpeername()* functions from glibc. Both will lead to the execution of the same *inet_getname* function on kernel side, though. Backward compatibility must be preserved in the glibc API and yet allow it to return a different, named socket address when requested. By improving the semantic of glibc functions, it is possible to distinguish which type of address should be returned. This is accomplished using the *namelen* (a pointer to an integer) input argument of those functions with different values: either the size of `sockaddr_in` or else the size of `sockaddr_in_named`, for IPv4. Inside the kernel, *inet_getname* is improved to take this into consideration.

5.2.3 Name→number mappings

It must stay clear that the model does not delegate all TCP rules to DOI Resolvers. Looking at the general TCP philosophy, we can conceive two components: (i) service allocation and (ii) name resolution and validation procedures. The first is dealt inside the server side Linux kernel and consists on allocating several resources including an address: a port number that cannot conflict with another (validated in *inet_csk_get_port*). The latter is handled by DOI Resolvers on both hosts. Therefore, except minor semantic changes, the same mechanism is applied as in section 5.1.2. A consequence is that using the same port name will, for all purposes, be considered as binding exactly the same service, leading to a failure when multiple instances make use of it.

5.2.4 Port names in TCP segments

Each client will use a different identifier for each connection. An integer is incremented in the modified *tcp_v4_connect* function and becomes part of the port name. As this identifier is used to correlate SYN and SYN+ACK segments, instead of the destination port number, 32 bit were allocated for this purpose which should be enough for dealing with multiple connection attempts from the same endpoint. Port names are actually structures with a special semantic identified by the associated DOI. It was considered a maximum of 256 different semantics. The definition of the enhanced port name structure and an example are shown in Table 5.2.

id (4 bytes)	doi (1 byte)	portname_data (x bytes)
0x00000001	0x00	F0306A$\cdots$

Table 5.2: Definition (*id, doi and portname_data*) of port names in the enhanced model context, followed by an example.

[1] A temporary socket used before establishing the connection to act on the connection request

Although the Linux TCP stack does not handle the exchange of user data in SYN and SYN+ACK synchronization segments, data in a named ACK after a SYN+ACK will be processed as standard payload. This was confirmed to happen both in MS Windows and Linux based operating systems. As it is mandatory to prevent port names from reaching upper layers, *tcp_data_queue* was modified to ignore the port name payload on named ACK segments.

5.2.5 Major kernel changes

The main changes and enhancements implemented in the Linux kernel involved:

- making the processing of *three-way handshake* segments asynchronous;
- changing some function definitions to pass and deal with more arguments;
- creating auxiliary functions, some for asynchronous handling and other to interact with DOI.

A detailed description of changes made within each kernel file can be found in Appendix D; some are worth mentioning, though.

The kernel routines *local_bh_disable* and *local_bh_enable* are used to disable and enable software interrupts on the local CPU, respectively. The functions *bh_lock_sock*/*bh_lock_sock_nested* and *bh_unlock_sock* serve to hold or release a lock on the socket. This lock is used to avoid changes in the state of the socket, due to incoming packets, interrupts or any handler. All of them are important and must be considered when the processing of a segment is broken due to interaction with DOIs.

Whenever a SYN segment is received, *tcp_v4_rcv* is interrupted and *tcp_prot.resolve_portname* is invoked in order to send a resolution request to the appropriate DOI Resolver. When the resolution comes, the *tcp_async_synack* continues the remaining segment processing. The standard SYN+ACK segment handling is interrupted in *tcp_rcv_synsent_state_process* which then invokes *tcp_prot.validate_portname* to send the validation request to the associated DOI. Its reply will invoke the kernel code of *tcp_async_ack* that will either send a named ACK or an RST. The named ACK segment will in turn be revalidated by the server when *tcp_v4_do_rcv* invokes *tcp_check_req_lite* (a simplified "clone" of *tcp_check_req*) that calls the *tcp_prot.validate_portname* function defined within the DOI interface kernel module.

A subtle difference to SRM concerns the hashing algorithm that calculates the index used by socket hashed lists. In ERM the length of the port name was not considered since port names in named SYN and SYN+ACK segments are not the same anymore. Instead, function *inet_ehashfn* was modified to use the port name's identifier field (*id*).

Concerning TCP windowing, the Linux kernel source code states that some endpoints have problems whenever TCP windows superior to 32767 are used (no scaling). Therefore, *MAX_TCP_WINDOW* is defined with this value. Even if 32767 is less than the 65535 allowed window size, besides being enough it is also a safe value that can and was chosen to be advertised by both endpoints, without needing any window scale.

5.2.6 DOI interface kernel module

As soon as the DOI interface kernel module (DOI LKM is loaded, an initialization function is invoked, as shown in *code snippet* 1. The module starts by invoking *netlink_register_notifier*

which registers a notifier to handle asynchronous Netlink events (in this case it just ignores them).

Afterwards, it creates a Netlink socket through *netlink_kernel_create*. Several parameters must be passed, though. One of them is the protocol type, an integer that indicates a kind of channel to be used for some specific purpose (e.g. routing table and firewall manipulation), between kernel and user processes. For this work, it was used the *NETLINK_TEST* macro, with the assigned and unallocated value 17, to distinguish the proposed protocol for DOI message exchanges. Another input parameter is a pointer to a function responsible for asynchronously handling the incoming Netlink datagrams, *nl_data_ready* more precisely.

Next, some linked lists are initialized, like *doihash* which contains the PID[2] of the user process for every registered DOI Resolver.

Finally, the module must somehow couple with the TCP stack and present the new available features. This is done by dynamically redefining some static variables inside the kernel. These variables are in fact function pointers, which point to NULL by default. As an example, the expression "*tcp_prot.register_portname=register_portname*" substitutes the *register_portname* in the *tcp_prot* kernel structure with a pointer to the *register_portname* defined locally inside the module.

When the module unloads, a similar process is done but in the reverse way. Note that the module sends a protocol message to all registered DOI Resolvers informing them of this event, a sort of graceful close (see *code snippet* 2).

Incoming packets are handled asynchronously, as mentioned earlier. Part of that processing is shown in *code snippet* 3. Basically, the datagram is copied into a *sk_buff* structure, its message type is analyzed accordingly with the DOI protocol messages and interpreted correspondingly. In the source code, it is visible part of the procedure that occurs when a DOI Resolver registers with the kernel module. The work flow for this use case is summarized as follows:

1. a datagram is received and copied into some reserved space;
2. the message type is analyzed;
3. the DOI's identifier (*doi*) is validated to check if no other DOI Resolver is already responsible for that ID (not shown);
4. a positive or negative acknowledgment is sent back to the DOI Resolver candidate;
5. in case of success, the module informs the DOI Resolver of all registered TCP endpoints for that domain of interpretation, if any.

One of the important features that permits some fault-tolerance is precisely the last point. If for some reason the DOI Resolver disappears and comes back online, there is no incoherence in the list of listening endpoints between the TCP stack and the DOI Resolver.

For every kernel "broken packet handling" or initiated interaction there must be its module counterpart, implemented as module functions. The list of kernel functions overridden by the kernel module are namely:

register_portname Registers a port name within the DOI. It is called whenever a TCP service starts listening; the DOI LKM will send a registration request to the appropriate

[2]Process ID

Code Snippet 1 DOI LKM initialization function used upon module loading.

```
static int __init my_module_init(void)
{
        int error;
        error = netlink_register_notifier(&netlink_notifier);
        if (error){
                printk(KERN_ERR "%s: register of event notifier failed: %d\n",
                        __FUNCTION__,error);
                return(1);
        }

        nl_sk = netlink_kernel_create(NETLINK_TEST, 0,
                                      nl_data_ready, NULL, THIS_MODULE);
        if (!nl_sk) {
                printk(KERN_ERR
                       "netlink_test: unable to create netlink socket!\n");
                return(1);
        }

        printk(KERN_INFO "netlink_test: initializing DOIHASH...\n");
        doihash=kmalloc(sizeof(struct doihash_element),GFP_KERNEL);
        INIT_LIST_HEAD(&doihash->list);
...
        tcp_prot.register_portname=register_portname;
        tcp_prot.unregister_portname=unregister_portname;
        tcp_prot.resolve_portname=resolve_portname;
        tcp_prot.validate_portname=validate_portname;
        tcp_prot.revalidate_portname=revalidate_portname;

        return 0;
}
```

DOI Resolver; A pointer to the listening socket and the corresponding listening TCP port are also passed as arguments (their use depends on the DOI logic);

unregister_portname Similar to *register_portname* with two exceptions: (i) it is invoked when the socket is destroyed (in *tcp_unhash*) and (ii) the message sent to the DOI Resolver corresponds to a deregistration;

resolve_portname Resolves an incoming connection request, when a SYN is received, during the *three-way handshake*. Arguments include the destination IP address and TCP port along with a pointer for the packet saved within the *sk_buff* structure. The latter will be used to distinguish and ignore duplicate requests coming from the kernel;

validate_portname Validates, in the client side, the "resolution" returned with SYN+ACK during the *three-way handshake*. Its arguments include the original, unresolved port name sent with the SYN and the resolved port name returned with the SYN+ACK.

revalidate_portname Re-validates, in the server side, the "resolution" returned with ACK segment during the *three-way handshake*.

Code Snippet 2 DOI LKM function used upon module unloading.

```
static void __exit my_module_exit(void)
{
        struct list_head *pos, *q;
        struct doihash_element *tmp;
        list_for_each_safe(pos,q,&(doihash->list)){
                tmp=list_entry(pos,struct doihash_element,list);
                printk(KERN_INFO "SF: freeing (doi=%i,pid=%hu)\n",
                        tmp->doi,tmp->pid);
                send_doi_msg(nl_sk,tmp->pid,++request_id,
                            DOI_MSG_TYPE_UNREGISTER_OK,0,NULL);
                list_del(pos);
                kfree(tmp);
        }
...
        tcp_prot.register_portname=NULL;
        tcp_prot.unregister_portname=NULL;
        tcp_prot.resolve_portname=NULL;
        tcp_prot.validate_portname=NULL;
        tcp_prot.revalidate_portname=NULL;

        if ((nl_sk) && (nl_sk->sk_socket)){
                sock_release(nl_sk->sk_socket);
                netlink_unregister_notifier(&netlink_notifier);
        }
}
```

Worth mentioning is that these functions do not return any data, since they are asynchronous. The "responses", or resolutions, are sent by DOI Resolvers using the DOI protocol (see section 5.2.8). Then, they are interpreted by the DOI LKM and a kernel function is asynchronously invoked finishing up the packet processing, that started with the resolution request but had to be broken due to the DOI mechanism. The kernel functions called from within the LKM, upon resolution request reception, are:

tcp_async_synack Continues the processing of an incoming SYN. Called in response to *resolve_portname*;

tcp_async_ack Continues the processing of an incoming SYN+ACK. Called in response to *validate_portname*;

tcp_async_ackchk Continues the processing of an incoming ACK during the *three-way handshake*. Called in response to *revalidate_portname*;

5.2.7 DOI Resolvers

The functionalities of two different DOI Resolvers were implemented in C language, within the same program named **doi_example**. The structure of the DOI Resolvers is basically the same so a simple command line argument can be used in order to select either model. Besides choosing which resolver to use, it is possible to change (see Listing 5.1) some parameters or even force in some way the behavior of the resolver.

Code Snippet 3 DOI LKM handling procedure for incoming Netlink datagrams.

```
static void nl_data_ready (struct sock *sk, int len)
{
...
receive_doi_msg(nl_sk,&skb,&err);
if (nlh->nlmsg_type==DOI_MSG_TYPE_REGISTER){
    struct doihash_element *node=kmalloc(sizeof(struct doihash_element),
                                         GFP_ATOMIC);
    node->doi=doi;
    node->pid=pid;
    list_add(&(node->list),&(doihash->list));

    send_doi_msg(nl_sk,pid,id,DOI_MSG_TYPE_REGISTER_OK,0,NULL);

    struct regnamehash_element *riter;
    struct doi_msg_register_portname rp;
    list_for_each_entry(riter,&(regnamehash->list),list){
            if (riter->doi==doi){
                rp.resolved_port=riter->resolved_port;
                rp.resolved_portname_len=riter->resolved_portname_len;
                send_doi_msg_with_header(nl_sk,pid,++request_id,
                    DOI_MSG_TYPE_REGISTER_PORTNAME,
                    sizeof(struct doi_msg_register_portname),&rp,
                    riter->resolved_portname_len,riter->resolved_portname);
            }
    }
...
```

Syntax:
- **−m** : use Exact Match DOI resolver;
- **−e** : use Regex DOI resolver;
- **−s** : force SYNACK resolve failure upon receiving a named SYN;
- **−a** : force ACK validation failure upon receiving a named SYNACK;
- **−c** : force ACK revalidation failure upon receiving a named ACK;
- **−d doi** : DOI integer identifier − *"doi"*;
- **−i** : startup id used for the Netlink DOI protocol's registration message;
- **−x name** : forced port name data sent in a named SYN − *"resolved_portname"*;
- **−y name** : forced port name data sent in a named SYN/ACK − *"acked_portname"*;

Listing 5.1: DOI Resolver command line options - syntax and description

A DOI Resolver behaves like a server, so it processes requests and sends the results back to the client which in this case is the kernel module. The first consequence is that the DOI Resolver can be stopped at any time, so there must be some way to provide this feature. As *code snippet* 4 shows, during the startup of the program a signal hander is defined for SIGINT, the signal that corresponds to an interrupt from keyboard (e.g. CTRL+C). The handler just changes a global control variable, which is not just enough because the program can and certainly will be blocked on some system call (e.g. waiting for incoming data). This can be

changed, though; through *sigaction* it's possible to modify the behavior of the signal handling whenever a signal is received on current blocked system calls initiated by that process. The flag *SA_RESTART*, if disabled, can make systems calls abort and set the *errno* global variable with the value EINTR.

The next step consists on sending the registration message to the kernel module using the Netlink socket, created at the program beginning.

Code Snippet 4 DOI Resolver initialization detail.

```
void leave(int sig){
        printf("received SIGINT signal...\n");
        out=1;
}

void main(int argc,char *argv[]) {
 while ((c = getopt (argc, argv, "mesacx:y:")) != -1)
         switch (c){
  ...
 sock_fd = socket(PF_NETLINK, SOCK_RAW,NETLINK_TEST);

 (void) signal(SIGINT,leave);   // an INTERRUPT signal is dealt by leave()
 struct sigaction oldact;
 int ret=sigaction(SIGINT, NULL,&oldact);
 // abort running system calls if a signal is received
 oldact.sa_flags&= ~SA_RESTART;
 ret=sigaction(SIGINT, &oldact,NULL);

 // register this DOI resolver in the kernel module
 send_doi_msg(sock_fd,pid,id++,doi,DOI_MSG_TYPE_REGISTER,0,NULL);
...
```

The resolver life cycle consists of consecutively answer to requests coming out from the kernel, through the module. This translates to a loop like the one presented in *code snippet* 5. Every cycle starts by being "blocked" on a instruction that sits waiting to receive a datagram from the LKM. Data is read onto an *iovec* structure that contains the Netlink header along with the DOI message. After finding out the DOI protocol's message type, it can be processed. In the given example, after receiving the port name registration message (*DOI_MSG_TYPE_REGISTER_PORTNAME* message type) the DOI Resolver will add the port name to a linked list of registered port names, which in the end correspond to a list of running TCP services.

The resolution request procedures are depicted on *code snippet* 6. The list of registered names must be iterated in order to find, if any, the pretended one. This logic is not mandatory, it was a decision taken given the two DOI Resolver models implemented. If some kind of hashing code was implemented and the port name contained that info, then the name lookup could be more direct and efficient. Anyway, each registered name is "compared" against the SYN connection request's port name (*name1*). In the example this is accomplished by *match* function. Finally, an answer is built with the hopefully resolved port name and sent back to the kernel module.

Code Snippet 5 DOI Resolver processing loop.

```
while(!out){
 int received_bytes=0;
 do{
        received_bytes=receive_doi_msg_flags(sock_fd,&msg,MSG_WAITALL);
 } while ((((received_bytes<=0) && (errno!=EINTR)) || (errno==EAGAIN))
            && (!out));
 if ((out) || (errno==EINTR)){  // if aborted
        break;
 }

nlh=((struct iovec *)(msg.msg_iov))->iov_base;
unsigned int request_id = ((struct doi_msghdr *)NLMSG_DATA(nlh))->id;
unsigned char msg_type = nlh->nlmsg_type;

 switch(msg_type){
    case DOI_MSG_TYPE_REGISTER_PORTNAME:
        struct doi_msg_register_portname *rp=NLMSG_DATA(nlh)+
                    sizeof(struct doi_msghdr);
        portname=NLMSG_DATA(nlh)+sizeof(struct doi_msghdr)+
                    sizeof(struct doi_msg_register_portname);
        void *portnameptr=malloc(rp->resolved_portname_len);
        memcpy(portnameptr, NLMSG_DATA(nlh)+sizeof(struct doi_msghdr)+
          sizeof(struct doi_msg_register_portname), rp->resolved_portname_len);
        /* add listening socket to list */
        append(&blist, rp->resolved_port, rp->resolved_portname_len,
                    portnameptr);
        break;
...
```

Exact Match DOI Resolver

The exact match DOI Resolver is similar to the Simple TCP name resolver, in the way that the concept behind it is the same. A client connects and proposes a name sent within the TCP port name option. The server side DOI Resolver, for the domain specified in the request, "iterates all registered TCP names" and tries to make an exact match with the port name they registered upon the socket binding procedure. As the equality match is in fact a byte equality, the *memcmp* function is used to make the comparison. The name resolution is rather simple, as *code snippet* 7 shows.

Since this model is an exact match resolver, the resolved port name and the name proposed in the SYN are effectively the same. Therefore, the SYN+ACK segment does not need to transport the resolved name. The named SYN+ACK and ACK segments must transport the port name header, so they are delivered to the right DOI for finishing up the name handling.

REGEX DOI Resolver

The REGEX DOI Resolver is a possible evolution of the previous DOI Resolver. Instead of doing a byte equality comparison, this resolver uses a regular expression specified in the SYN's port name *name1*. So, a service is found if its registered name matches the given regular

Code Snippet 6 DOI Resolver *resolve port name* request handling.

```
...
struct doi_msg_resolve_answer ans;

found=0;
list_for_each_entry(iter,&blist.list,list) {    // iterates list
    if (using_exact){
            if ( exactmatch(iter->resolved_portname+sizeof(struct portnamehdr),
                portname+sizeof(struct portnamehdr),iter->resolved_portname_len,
                portname_len) ){
                    found=1;
                    break;
            }
     } else if (using_regex){
            if ( regexmatch(iter->resolved_portname+sizeof(struct portnamehdr),
                portname+sizeof(struct portnamehdr)) ){
                    found=1;
                    break;
            }
     }
}
if (found){
    // force the same ID, so the kernel can correlate requests
    ((struct portnamehdr *)iter->resolved_portname)->id = \
                                        ((struct portnamehdr *)portname)->id;
    // build answer message
    ans.resolved_port=iter->resolved_port;
    ans.resolved_portname=iter->resolved_portname;
    ans.resolved_portname_len=iter->resolved_portname_len;
} else {
    ans.resolved_port=0;
    ans.resolved_portname=NULL;
    ans.resolved_portname_len=0;
}
send_doi_msg(sock_fd,pid,request_id,doi,
    DOI_MSG_TYPE_RESOLVE_ANSWER,sizeof(struct doi_msg_resolve_answer),&ans);
...
```

expression The matching function is seen in *code snippet* 7. GNU C library, glibc[3], comes with standard Linux distributions and supports the POSIX.2 interface, namely the POSIX regular expression compilation. Therefore, for simpleness, this library's functions were used although other alternatives exist such as PCRE[4] for a Perl 5 regular expression like syntax.

The model here presented resolves the proposed textual port name to some other name. It might make sense to return the resolved name in this specific scenario. As such, the named SYN+ACK segment contains the TCP port name option with the resolved port name. On response, the client issues an ACK segment which transports only the port name header, with the correct *id* and DOI identifier - *doi*.

[3] `http://www.gnu.org/software/libc/`
[4] `http://www.pcre.org`

Code Snippet 7 Function used to resolve the name in the *Exact Match* DOI Resolver.

```
int exactmatch( char *name1, char *name2, unsigned short int len1,
                                          unsigned short int len2) {
  if((len1!=len2) || (memcmp(name1,name2,len1)!=0)) {
          return 0;
  } else {
          return (1 && !force_synack_reject);
  }
}
```

Code Snippet 8 Function used to resolve the name in the *REGEX* DOI Resolver.

```
int regexmatch(const char *string, char *pattern) {
  int status;
  regex_t re;

  if(regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0)
        return 0;

  status = regexec(&re, string, (size_t)0, NULL, 0);
  regfree(&re);
  if(status != 0)
        return 0;
  return (1 && !force_synack_reject);
}
```

5.2.8 DOI Protocol

The DOI protocol allows data transactions between the TCP stack, running in kernel space, and the DOI Resolvers, running on user space. The requisites for a DOI protocol are:

1. having an efficient kernel space ↔ user space protocol;
2. providing a mechanism for asynchronous messages (connectionless protocol);
3. being a reliable protocol;
4. supporting addressing features;
5. supporting buffered, queued messages.

There are several alternatives for kernel space ↔ user space communication.

System calls provide a standard interface to access kernel functions but adding a new one breaks portability and introduces security concerns.

Another option could be using **ioctl** or **sysctl** interfaces. The *ioctl* function executes commands on opened file descriptors (i.e. stream devices) while *sysctl* permits reading or realtime adjustments on system's internal parameters.

Linux also supports the ***/proc***, or the more recent ***/sys***, virtual file system (VFS). The set of directories and files stored within this FS[5] map to a kernel view of the system, where

[5]File System

each device, driver or some parameter is seen as some file(s) in a tree like hierarchy. In a kernel module it's possible to add entries to that VFS and to handle operations for that *files*, like *read/write*. Although interesting, because it allows an almost transparent way for user processes to interact with the kernel, it completely lacks addressing and it has no decent security, except the one provided by the filesystem itself.

Netlink [35] is a kernel↔user level communication protocol that uses the socket metaphor to provide an asynchronous API, for connectionless data exchange. Besides unicast, that uses PID's as addresses, Netlink also supports multicast using a 32 bit mask to choose the destination multicast groups. There are API's to ease the kernel side but also the user side implementations, even if they are a bit different.

The developed protocol is based in Netlink sockets, since it fulfills the initial requisites and resembles a typical communication protocol. The Netlink addressing scheme considered is unicast, because datagrams are sent from the LKM to a specific DOI Resolver and from each DOI Resolver to the LKM, in the opposite direction. The destination address is defined as a *sockaddr_nl* structure where the *nl_pid* specifies the destination *address*. When the datagram's destination is the kernel, the PID must be equal to "0". If it is the DOI Resolver then it must have the value of its PID. The multicast group defined by *nl_groups*, given the fact that it is not used, must be zero in either direction.

```
struct sockaddr_nl
{
  sa_family_t    nl_family;  /* AF_NETLINK   */
  unsigned short nl_pad;     /* zero         */
  __u32          nl_pid;     /* process PID */
  __u32          nl_groups;  /* multicast groups mask */
} nladdr;
```

Messages exchanged using Netlink must be preceded by a *nlmsghdr* header. The *nlmsg_len* defines the total length of the packet, whose payload size is fixed and defined by a global variable *MAX_PAYLOAD* equal to 1024 (bytes). The *nlmsg_type* and *nlmsg_seq* exist to help and be used directly by the "Netlink protocols" themselves. The DOI protocol has a set of different messages, so the message type can be differentiated through the *nlmsg_type* field. The same applies for the sequence number *nlmsg_seq* that can be used by the protocol message to establish some correlation between requests and replies, for example. The *nlmsg_pid* field follows the same semantics as in *sockaddr_nl* except that is now applies to source address.

```
struct nlmsghdr {
    __u32 nlmsg_len;    /* Length of message including header. */
    __u16 nlmsg_type;   /* Type of message content. */
    __u16 nlmsg_flags;  /* Additional flags. */
    __u32 nlmsg_seq;    /* Sequence number. */
    __u32 nlmsg_pid;    /* PID of the sending process. */
};
```

The protocol messages themselves, referred as "doi_msg", are composed by an header (*doi_msghdr*) and a payload (*doi_payload*). The header has a packet identifier (*id*), a DOI identifer (*doi*) and a field to tell the size of the payload - *payload_len*. The *nlmsg_seq* could be reused instead of defining an *id*, since they can provide the same functionality. Since the DOI protocol is layered on top of Netlink messages, it was decided to keep the different semantics even if both variables are fulfilled with the same values.

```
struct doi_msghdr{
        unsigned int            id;
        unsigned char           doi;
        unsigned short int      payload_len;
} doi_msghdr;
```

Finally, after the DOI message header, follows the payload containing the request or reply data. Although Netlink messages are asynchronous, from the DOI protocol point of view the DOI messages follow a "request→reply" philosophy. Therefore, the messages can be grouped as Table 5.3 shows.

Request Message	**Reply Message**	**Notes**
doi_msg_register_portname	NONE (the reply message does not need any reply payload data; the success/-failure is directly obtained by the message type)	Used when registering and unregistering port names, respectively after a bind() and socket destruction.
doi_msg_resolve_request	doi_msg_resolve_answer	First resolution request and answer, after named SYN segment.
doi_msg_validate_request	doi_msg_validate_answer	Validation request and answer, after named SYN/ACK segment.
doi_msg_revalidate_request	doi_msg_revalidate_answer	Re-validation request and answer, after named ACK segment.

Table 5.3: DOI protocol messages (request and consequent reply) used for data exchange between the DOI LKM and the DOI Resolver.

In Appendix D.1 the different protocol message structures are detailed. Every DOI request can succeed or fail, i.e. the validation logic may decide to proceed with the connection or to abort it. This is signaled through a variable named *result*, where a "0" indicates an abort and a "1" tells the operation had success or the connection should proceed, depending on the context.

A global view or port name encapsulation is shown in Fig. 5.3. One or more *portname*'s, each one composed by an header and data, possibly with some additional data are all encapsulated in a DOI message. The structure of the latter one depends on the message type, identified in the Netlink message header. Note that while the *doi_msghdr*'s *id* field has the same value as *nlmsg_seq*, the *portname*'s *id* means totally a different thing - it is the integer used for socket correlation during the *three-way handshake*.

For the purpose of abstracting the underlying protocol details, a set of functions were developed as seen in Appendix E. They masquerade the API differences found in Netlink sockets between kernel and user processes.

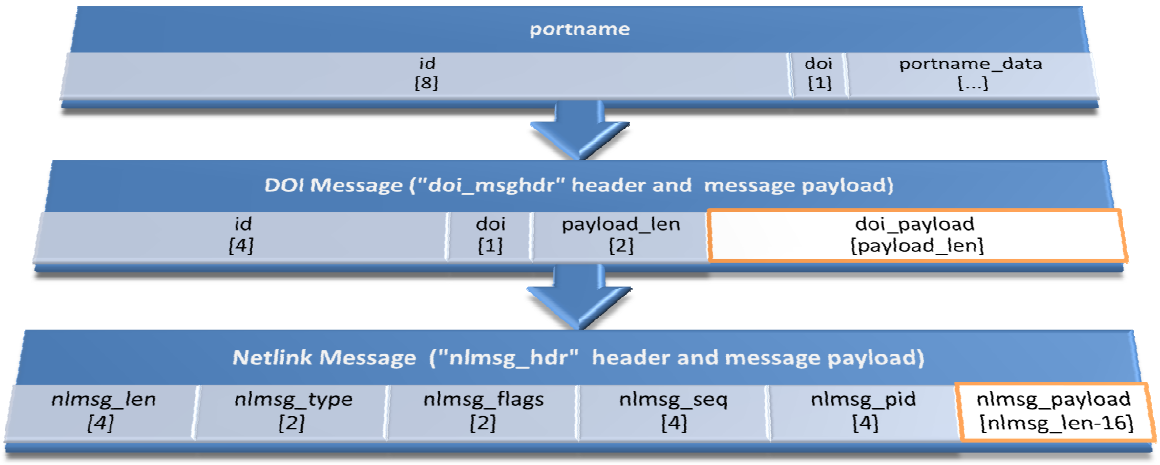

Figure 5.3: Global view of port name encapsulation, in several layers, starting with DOI messages and their respective encapsulation in Netlink messages.

Chapter 6

Evaluation

In order to evaluate the features described in Chapter 4, some basic TCP client/server programs were implemented using the new `sockaddr_in_named` structure. They ran in a server with the modified Linux kernel, as mentioned in Chapter 5. Some well known applications were also patched, like the Apache2 web server and Netcat utility. Apache2 was partially patched to bind to a specific port name. Fedora 7 Netcat source package was used as a basis to build a command line application supporting both port number and port name bindings.

```
Syntax:
 −b bindmode : Bind mode (”0”−port only; ”1”−port name only; ”2”−port+port name)
 −o doi      : DOI integer identifier , used within the port name header;
 −P portname : port name data, defined as a null terminated string;
 −L psize    : port name total size , including port name header and data
               (null padded if needed);
```

Listing 6.1: New command line options syntax for enhanced Netcat

As security is delegated to DOI Resolvers, there were no specific tests to address this neither the time to implement a DOI Resolver covering authentication algorithms.

6.1 Conformity testing

Conformity tests consisted in the evaluation of two attributes: correctness and transparency. Correctness means that introduced changes must be correct, according with the standards. Transparency means that changes or features introduced in the proposed models must be transparent to current TCP stacks. In other words, already deployed solutions must interact correctly with the modified systems, without having to change the first ones. From a macro view, this translates into four items:

1. correct TCP connection establishment;
2. RST segment when a named connect is issued to a legacy TCP stack;
3. interaction with firewalls;
4. interaction with NAT boxes.

For SRM, all possible combinations of server port name bind modes with client connection methods were tested successfully within and between machines with the standard and the modified TCP stack, as shown in Table 6.1. Clients with old TCP stacks can only connect to port numbers (cases 3,4) and servers with old TCP stacks can only handle connection requests including a valid port number (cases 2,4). Clients with new TCP stack (cases 1,2) can either connect by port number or name but the latest will only be fully understood by the new port name aware TCP stack (case 1).

		Server-side stack	
		new	old
Client-side stack	new	✓[1]	✓[2]
	old	✓[3]	✓[4]

Table 6.1: Interoperability between current (old) TCP stacks and the new proposed stacks, implementing either the Simple or Enhanced TCP Name Resolution models.

When using ERM, successful named connections can only be established between endpoints with the new TCP stack and a common DOI (case 1). A named SYN, although conforming with TCP, will not be processed as expected by the server endpoint (case 2). Legacy connections can occur between any type of stack, even if they are intended to servers following the ERM architecture (cases 3,4).

The integration with current firewalls and NAT boxes was evaluated by the following tests.

Firewall Test [SRM,ERM] The proposed architecture should integrate with firewalls.

Procedure:

1. Two hosts, H_1 and H_2, should be initially configured for the SRM scenario;
2. in H_2, a TCP server is bound to the name "test" and port number 12000;
3. in H_2, run and configure iptables as a firewall, enabling access only to TCP port 0:

```
# Accept from all to ports do TCP port 0
iptables -A INPUT -p tcp --dport 0 -j ACCEPT
#iptables -A INPUT -p tcp --dport 12000 -j ACCEPT
# drop remaining traffic
iptables -A INPUT -j DROP
```

4. issue a connection request from H_1 to H_2, specifying the port name data "test";
5. repeat the connection request, enabling access only to TCP port 12000;
6. repeat the connection request, enabling access to both TCP ports 0 and 12000;
7. repeat the tests using ERM in hosts H_1 and H_2;

Expected Result: The first connection request will fail because the ACK segments uses the resolved port instead of 0. The second connection attempt will fail since the named SYN ist sent to the unallowed TCP port 0.

Result: Ok, it is possible to establish a connection only when both ports 0 and the resolved one are allowed in the firewall. The output of *conntrack*[1] shows that iptables sees two connections: (i) one in SYN_RECV state that corresponds to the received SYN

[1]A LKM for maintaining connection state tracking. Its state can be seen in */proc/net/nf_conntrack* file.

and (ii) another one in ESTABLISHED state caused by the last two segments that use the resolved port - SYN+ACK and ACK.

NAT Test [SRM,ERM] The proposed architecture should integrate with NAT boxes, namely whenever using source IP masquerading (e.g. by routers) and port forwarding (e.g. by gateways).

Procedure:

1. Two hosts, H_1 and H_2, should be initially configured for the SRM scenario;
2. in H_2, a TCP server is bound to the name "test" and port number 12000;
3. in H_1, run and configure iptables to perform source masquerading:

```
# accept incoming traffic sent to localhost
iptables -A INPUT -i lo -p all -j ACCEPT
# accept outgoing traffic from localhost
iptables -A OUTPUT -o lo -p all -j ACCEPT
# accept incoming traffic from connections already established
iptables -A INPUT -i eth0 -m state --state  ESTABLISHED,RELATED -j ACCEPT
# perform source masquerading
iptables -o eth0 -t nat -A POSTROUTING -j MASQUERADE
# drop remaining traffic
iptables -A INPUT -j DROP
```

4. issue a connection request from H_1 to H_2, specifying the port name data "test";
5. using another host, H_G, run and configure iptables so it behaves as a gateway, forwarding traffic from H_1 to H_2 (e.g. 192.168.73.128) through H_G:

```
# enable IP forwarding
echo 1 > /proc/sys/net/ipv4/ip_forward
# accept incoming traffic sent to localhost
iptables -A INPUT -i lo -p all -j ACCEPT
# accept outgoing traffic from localhost
iptables -A OUTPUT -o lo -p all -j ACCEPT
# accept incoming traffic from connections already established
iptables -A INPUT -i eth0 -m state --state  ESTABLISHED,RELATED -j ACCEPT

# Accept from all to ports do TCP port 0
iptables -A INPUT -p tcp --dport 0 -j ACCEPT
# forward TCP/IP traffic to host 192.168.73.128 (H2)
iptables -t nat -A PREROUTING -p tcp --dport 0 -j DNAT --to 192.168.73.128
# perform source masquerading
iptables -o eth0 -t nat -A POSTROUTING -j MASQUERADE

# drop remaining traffic
iptables -A INPUT -j DROP
```

6. repeat the tests using ERM in hosts H_1 and H_2;

Expected Result: Source masquerading should not affect neither SRM nor ERM. Port forwarding will not work unless NAT boxes are updated.

Result: The output of *conntrack* shows that iptables sees one connection in the SYN_SENT state that corresponds to the forwarded SYN. The iptables on H_G does not correlate the named SYN+ACK sent by H_2 with the existing connection since the SYN+ACK TCP source port uses the resolved port instead of 0.

The previous tests depict some problems with existing network elements that must be updated for a smooth integration with the proposed models as have been presented. Both

firewalls and NAT boxes suffer from the "problem" of identifying connections based on source and destination IP addresses plus TCP port numbers.

Several correctness and compatibility issues were discussed along an intense and fruitful discussion with Dr. Joseph (Joe) Touch, Director of the Postel Center[2], active member of "TCP Maintenance and Minor Extensions Working Group" and editor of several RFCs. The discussed topics are in essence here summarized.

Port changes during connection establishment Many applications and network equipments identify connections by ⟨*IPsrc, TCPsrc, IPdst, TCPdst*⟩ like entries, as happens with NAT boxes, firewalls (e.g. iptables), among many other. Changing port numbers during the connection establishment phase may therefore be problematic.

As mentioned earlier, NAT is used everywhere. NAT source masquerading should not be a problem since it only affects the source IP address and TCP port number. Port forwarding, applied at server side, might change both the destination IP address and TCP port number. For NAT to work in the proposed architectures, the NAT box would have to forward the resolution request to the target host (and maintain internally the forwarded port name), which would return a resolved port that would be used thereafter as the connection identifier.

IPSec [19] supports two operational modes: the transport mode and the the tunnel mode. The first authenticates/encrypts only the transport payload (e.g. TCP segment) while the latter does the same but for the original, complete IP datagram. In transport mode, a security association (SA) is established for each of the two opposite packet flows. SA is a logic concept that serves for tagging traffic ruled by the same security processing. A SA selector restricts or widens the range of traffic that applies to a given SA. They are used during lookup of SAD (security association database) entries. Transport ports **may** be used as SA selectors, which raise a problem if port numbers are changed during the connection. One would have to update SAD selection algorithms and policies for fully supporting port names.

TCP-AO, still being finalized, relies on connection identifiers based on IP addresses and TCP ports. These are incompatible with the proposed models unless TCP-AO is updated to work during the name resolution and afterwards.

A possible solution for avoiding the mentioned problems would be the one depicted in Joe Touch's Draft for port/name decoupling: choose a random destination TCP port defined initially by the client in the SYN segment.

Usage of a reserved port TCP port 0 usage is reserved, fact which IANA confirmed. Two solutions are possible to supersede this problem. One is by using a specific, IANA registered, port for the context of port names, although port numbers are assigned having in mind user services, not kernel features. This still does not solve the problem caused by changing the port number during the connection. The other solution consists on applying the same philosophy used by Joe Touch's *Portnames Draft*. This apparently solves some problems (e.g. NAT) while avoiding reserving one TCP port number for the purposes of TCP port names.

[2] `http://www.postel.org`

Practical considerations The usage of a TCP port name option consumes 4 bytes within the TCP header of segments exchanged during the connection establishment phase. This might restrict the use of further TCP options although it does not seem a big issue due to the reduced size of the option itself. More, in ERM the port name can contain additional data related with the connection (it could even be thought as a means to transport further TCP options and therefore provide a way to permanently remove the current restriction on the usage of TCP options).

In SRM there can be only one port name mapping to a certain port number, since the implementation considers a one to one mapping. This is is not a real limitation because in current TCP stacks there can be only one TCP server bound to the same port number.

Using large port names might pose some problems. Fragmentation occurs when segments are large enough [27]. If a fragment is lost, due to network issues, then the complete TCP segment is lost since it cannot be reassembled. This will lead to the retransmission of entire segments, thereby affecting connection establishment time during the *three-way handshake*. A similar problem occurs when large windows are used and SACK is not supported. A maximum MTU of 576 bytes ensures that fragmentation will not occur. Therefore, the size of port name should be limited to the MTU, minus the IP and TCP headers. The IP and TCP headers vary from 20 bytes up to 60 bytes, depending on protocol options. So, an optimistic value for the maximum port name size would be $576 - 20 * 2 = 536$ bytes while a pessimist would be $576 - 60 * 2 = 456$ bytes.

The memory usage was another questioned subject. First, because using large names might lead to fragmentation, so the TCP stack must cache the fragmented segments for a considerable period. Moreover, using large port names by itself consumes more memory during packet processing. However, current TCP stacks also demand more resources whenever they receive more packets. Nevertheless, the question is pertinent when the retransmission of segments during the *three-way handshake* is needed.

The port name, in the Simple Resolution architecture, must be somehow kept in memory by the client for matching the incoming SYN+ACK or RST segments with the named connection request. In the server side, the port name is just needed during the processing of the incoming SYN. This additional state informational is not required however in the Enhanced TCP Resolution model, since segments and their associated sockets are correlated by an $\langle id, doi \rangle$ pair of the port name header.

Concerning SYN Cookies[3], when used the hosts ignore the payload on SYN segments [11] in order to avoid DoS attacks, as the ones originated by SYN flooding. In fact, hosts with current TCP stacks are as vulnerable to SYN flooding attacks as the modified systems.

While SYN Cookies break the semantics of TCP connection establishment, in ERM that is preserved and DOI Resolvers can implement complex and even stateless authentication logic.

Need for TCP name resolution models TCPMUX advantages may be the fact that it is an already deployed technology, clearly identified by a reserved port number which is compatible with NAT, ultimately its main advantage.

[3] `http://cr.yp.to/syncookies.html`

The SRM advantages include the fact it uses just one connection while TCPMUX requires either two connections (one from client to TCPMUX and another from TCPMUX to the destination service) or else a connection (from client to TCPMUX) followed by some IPC mechanism (or a fork) to communicate/execute the pretended service. The fact that SRM does not need any user space service makes it more fault tolerant. Even in the ERM, part of the logic resides inside the DOI LKM, providing the ground basis for dealing with faulty DOI Resolvers.

TCPMUX also damages the semantics of TCP because connections towards it always succeed, even when the connection to the pretended service fails. It also limits the usage of some names ("help", "-", "+", "⟨CR⟩⟨LF⟩") due to its protocol. As all connections are directed to TCPMUX on TCP port number 1, this port can be caught by port scanners and therefore be a preferable attack target. The proposed models maintain the TCP semantics, do not limit port names content and are resistant to port scanners. While TCPMUX provides only a naming service, the suggested models extend that and can include things like authorization+authentication.

Joe Touch's Internet Draft (rev. "00") uses names to address servers but it limits the names to UTF-8 strings. An important advantage of Joe's model is being NAT friendly, since the connection identifier, formed by the source and destination IP addresses plus TCP port numbers, is kept stable during connections' lifetime.

During the discussion, Dr. Joseph Touch said a quite interesting phrase, when talking about introducing new features to standards (e.g. protocols) widely deployed. Quoting Dr. Joseph Touch,

"That's an uphill battle in the IETF, and would require that you show something broken in TCP that everyone needs. It's not sufficient to show a feature that some could use."

which should be a major concern when thinking on protocol modifications, either due to changes or to new functionalities.

6.2 Functional testing

The list of functional tests served to validate the new functionalities introduced by the proposed models and are summarized in the following items.

1. pluggable mechanism to extend current, standard TCP stack;
2. pluggable DOI Resolvers;
3. registration and unregistration of port names, when a TCP server binds and starts listening or when it stops;
4. re-registration of port names, of current listening TCP servers, when DOI Resolvers crash/stop and reconnect;
5. correct interruption on the packet receiving logic, resulting in the invocation of the appropriate DOI functions (resolve, validate, revalidate)

6. correct correlation of sockets, using the *id* in the port name header, instead of TCP port number (along with IP addresses);
7. connection reset whenever DOI Resolver invalidates the operation, upon reception of named ACK, SYN+ACK and ACK segments;
8. continuation of the connection procedure whenever DOI Resolver validates the operation, in each step;
9. TCP stack independence between port names in ACK, SYN+ACK and ACK segments;
10. correct "routing" of TCP port names to the correct DOI;
11. use of large port names;
12. RST segment with port name, after a failure in a named connection request to a server with the modified TCP stack;
13. RST segment with no port name, after a failure in a legacy connection request to a server with the modified TCP stack;
14. bind mode behavior;

For the purposes of the following tests, TCP_x means TCP server "x", H_x means host "x", DOI_x corresponds to DOI Resolver "x". Tests 1...10 are applicable only to ERM while tests 13...14 are to be executed in the SRM environment. The tests 11 and 12 are intended for both models. The detailed tests, their procedures, expected and obtained results are now presented.

Test 1 [ERM] The proposed architecture should integrate with the existing TCP stack via a pluggable mechanism.

Procedure:

1. load in H_2, using *modprobe* command, the DOI LKM;
2. in both hosts, start a DOI Resolver with exact name matching for the domain's ID "0";
3. in H_2, a TCP server (TCP_x) is bound to the name "test";
4. issue a connection request from H_1 to H_2, specifying the port name data "test";
5. unload in H_2, using *rmmod* command, the DOI LKM;
6. repeat the fourth step;
7. repeat the first and fourth steps, in this sequence;

Expected Result: The TCP name service hook (LKM) can be added or removed on demand.

Result: Ok, only the second connection attempt fails since the DOI LKM is not loaded.

Test 2 [ERM] DOI Resolvers can be available within the new TCP stack via a pluggable mechanism.

Procedure:

- in H_2, a TCP server (TCP_x) is bound to the name "test1234";
- in H_2, a DOI Resolver with exact name matching is started for the domain's ID "0";
- issue a connection request from H_1 to H_2, specifying the port name data "test";
- in H_2, abort the previously running DOI Resolver and start a new one supporting regular expression name matching, for the domain's ID "0";
- repeat third step;

Expected Result: DOI Resolvers can be added or removed on demand.

Result: Ok, only the second connection attempt succeeds since "test" regular expression matches the bound name "test1234", which was a feature provided only by the second resolver.

Test 3 [ERM] Validate if TCP servers register/unregister name in the DOI, whenever they "start" or "stop" (listening).

Procedure:

- start a TCP server (TCP_x) binding it to the name "test", associated with DOI_x;
- stop TCP_x;

Expected Result: TCP servers register/unregister the port name in the appropriate DOI, whenever their socket state enters LISTEN or CLOSED.

Result: Ok, by analyzing the debug output of DOI_x it was verified the reception of the DOI protocol messages, in both cases.

Test 4 [ERM] Validate if DOI LKM re-registers port names whenever DOI Resolvers restart.

Procedure:

- a DOI Resolver DOI_x is started to handle requests for the domain's ID "0";
- start a TCP server (TCP_x) binding it to the name "test", associated with DOI_x;
- stop or kill DOI_x;
- start DOI_x again;

Expected Result: DOI_x should receive the port name registration after a restart.

Result: Ok, TCP servers are always accessible even after DOI Resolvers restart.

Test 5 [ERM] Validate if packet handling logic integrates with DOI LKM and DOI Resolvers.

Procedure:

- in H_1 and H_2, a DOI Resolver DOI_x is started to handle requests for the domain's ID "0";
- in H_2, start a TCP server (TCP_x) binding it to the name "test", associated with DOI_x;
- issue a connection request from H_1 to H_2, specifying the port name data "test";

Expected Result: The processing logic for incoming packets, previously synchronous, must be asynchronous, that is to say it gets suspended when DOI functions are invoked (resolve, validate, revalidate) and is resumed when DOI replies come back;

Result: Ok, the connection was established because DOI functions were invoked accordingly with the enhanced model, as DOI_x's debug output showed, and the packet handling integrated transparently with DOI;

Test 6 [ERM] Validate if sockets are correctly matched during the *three-way handshake*, using the port name header's *id* and the TCP port when appropriate.

Procedure:

- establish a named connection between two endpoints, where the fist SYN in sent towards TCP port 0;

Expected Result: The port name header's *id* is used to correlate a SYN with the SYN+ACK, instead of TCP ports (besides IP addresses). Afterward, TCP ports are used in the remaining segments.

Result: Ok, the connection was established successfully.

Tests 7,8 [ERM] Validate if *three-way handshake* proceeds normally, upon a positive acknowledgment from the DOI Resolver and if it's aborted upon a negative acknowledgment.

Procedure:

- in H_1, a DOI Resolver DOI_x is started and configured to return success/failure on validation requests;
- in H_2, a DOI Resolver DOI_y is started and configured to return success/failure on resolution and revalidation requests;
- issue connection requests from H_1 to H_2, for the multiple combinations of DOI_x and DOI_y configurations;

Expected Result: When DOI Resolvers are configured to return success on each operation, named SYN+ACK and ACK segments are sent back, after receiving a named SYN and SYN+ACK segments respectively. The connection is reset by sending back a RST segment, whenever DOI Resolvers return a negative acknowledgment.

Result: Ok, the connection could be aborted at any time depending only on the outcome of DOI Resolver's decision logic.

Test 9 [ERM] Validate if the port names can be distinct between named SYN, SYN+ACK and ACK segments.

Procedure:

- in H_1, a DOI Resolver DOI_x is started and configured to return success on validation requests besides forcing named ACK segments to contain port name data "something";
- in H_2, a DOI Resolver DOI_y is started and configured to return success on resolution and revalidation requests besides forcing named SYN+ACK segments to contain port name data "xpto";

- issue a connection request from H_1 to H_2, specifying the port name data "bla";

Expected Result: During the *three-way handshake* all port names can be different (it's up to the DOI to define their values).

Result: Ok, the connection was established and the success of *three-way handshake* depends only on the outcome of DOI Resolver's decision logic.

Test 10 [ERM] Validate if the port names are delivered to the right DOI.

Procedure:

- in H_2, two DOI Resolvers, DOI_x and DOI_y, are started for the domain's ID "0" and "1" respectively;
- issue two connection requests from H_1 to H_2, one for DOI "0" and another for DOI "1";

Expected Result: DOI protocol messages are delivered to the DOI identified by the port name header's *doi*.

Result: Ok.

Test 11 [SRM,ERM] Validate if the length of port names can be considerable large.

Procedure:

- In H_2, a ERM aware host, a DOI Resolver with the exact name matching logic is started;
- in H_2, TCP servers TCP_x and TCP_y are bound to port names whose sizes are 500 bytes and 4000 bytes, respectively;
- issue two connection request from H_1 to H_2, by specifying exactly the same port names as in the last point;
- repeat the second and third procedures in the SRM context;

Expected Result: Even if TCP segments suffer from IP fragmentation, they must be correctly handled.

Result: Ok, the connections were established successfully. Fragmentation was observed when using the port name of 4000 bytes.

Test 12 [SRM,ERM] Validate if a named RST, with the original port name, is sent back whenever a named connection request is made to an unavailable endpoint in a host with a modified TCP stack.

Procedure:

- issue a connection request from H_1 to H_2, specifying the port name data "test" (there must be no TCP server running in H_2 associated with the given port name);

Expected Result: A RST should be sent back, containing a port name similar to the request's port name.

Result: Ok. In SRM a named RST was sent with the port name "test". In ERM a named RST was sent with the port name's header fields $\langle id, doi \rangle$ equal to the ones used in the named ACK.

Tests 13,14 [SRM] Validate the correct behavior of TCP server's bind mode logic.

Procedure:

- in H_2, a TCP service (TCP_x) is bound to the port 12000 and name "test", with bind mode "port name only";
- issue a connection request from H_1 to H_2, specifying the TCP destination port 12000;
- in H_2, rebind (TCP_x) with bind mode "port + port name only";
- issue a connection request from H_1 to H_2, specifying the TCP destination port 12000 or the name "test";

Expected Result: A standard RST, with no port name, is sent back whenever a legacy connection request is made to a service with bind mode permitting only named connection requests.

Result: Ok, the first connection attempt returned a standard RST while the remaining connection requests succeeded.

6.3 DOI testing

The DOI tests are focused on the validation of the resolution logic implemented in the DOI Resolvers and also in the test of some of its features:

1. exact match logic;
2. regex resolver logic;
3. validate DOI registration;
4. DOI Resolver graceful close;

For the purposes of the following tests, H_x means host "x", DOI_x corresponds to DOI Resolver "x". The detailed tests, their procedures, expected and obtained results follow.

Test 1 Validate the resolution logic of the Exact Match DOI Resolver (DOI_x).

Procedure:

- in H_2, a TCP service (TCP_x) is bound to the name "test";
- issue a connection request from H_1 to H_2, specifying the port name data "test";
- validate if DOI_x resolves to the bound TCP_x service;
- issue a connection request from H_1 to H_2, specifying the port name data "tes";
- validate if DOI_x does not resolve to the bound TCP_x service;

Expected Result: The connection is established with success to TCP_x service in the first connection attempt.

Result: Ok, by analyzing the debug output of DOI_x it was verified the correctness of the resolution logic.

Test 2 Validate the resolution logic of the REGEX DOI resolver ($DOIr$).

Procedure:

- in H_2, a TCP service (TCP_x) is bound to the name "test123";
- in H_2, a TCP service (TCP_y) is bound to the name "test456";
- issue a connection request from H_1 to H_2, specifying the port name data "test4";
- issue a connection request from H_1 to H_2, specifying the port name data "test12";

Expected Result: The first connection is established to TCP_y server while the second one is established to TCP_x.

Result: The resolution logic was successful and TCP connections were established as expected.

Test 3 Validate the DOI registration logic to only allow one DOI Resolver per DOI.

Procedure:

- in H_1, a DOI Resolver is started to handle requests for the domain's ID (doi) "0" (there must be no DOI Resolver already running for that domain);
- in H_1, another instance of the DOI Resolver is started, also for the same doi;

Expected Result: Ok, only one DOI Resolver instance is able to register for a given domain ID.

Result: Ok, the first DOI registration succeeds but the second one fails.

Test 4 Validate that a DOI Resolver can be stopped at any time and that unregisters itself from the LKM, in a graceful close way.

Procedure:

- in H_1, a DOI Resolver DOI_x is successfully started for the domain's ID "0";
- in H_1, an abort by means of a SIGINT signal (e.g. a CTRL+C) is sent to DOI_x);
- in H_1, DOI_x is started once again for the domain's ID "0";

Expected Result: After the abort request, DOI_x sends immediately a deregistration message to the LKM and exits. After that, starting again DOI_x succeeds.

Result: Ok, DOI Resolver was promptly aborted, on user request, and thereafter it could be started successfully for the same domain.

Chapter 7

Conclusions and Future work

The main purpose of this work was to conceive and implement a secure TCP-layer name service. As mentioned in Chapter 1, many deployed TCP services are vulnerable to attackers due to two reasons: (i) they have bugs which sooner or later will be found and (ii) because access to them is freely allowed by the transport layer. Although firewalls can impose some accessing constraints, they are a foreign active element that only "blindly" allow or forbid the access to services and that do not negotiate any in-band security data which could be used to validate/authenticate the connection establishment. In fact, both TCP-layer name service and firewall mechanisms can and should coexist.

Two models were devised for a TCP name service, both by extending the TCP synchronization segments and thus not interfering with the normal data exchange during the connection. This conforms with the TCP standard and is compatible with existing TCP implementations.

The first presented model - Simple TCP Name Resolution - consisted of a simple name service for TCP ports. This name service allows TCP clients to identify services by a port name, instead of port number, which is more user-friendly. The simplicity of SRM and the fact that it can be directly integrated into the TCP/IP stack is also an important point. One security advantage of this addressing mechanism is that it allows services with unusual names, known only by small communities, to remain undetected by port scanners (i.e. security by obscurity).

The main disadvantage of SRM is due to its fixed logic, the lack of extensibility which restricts it to a mere name resolution methodology. No other features, as the ones concerning security by means of data exchange, can be built into it.

The Enhanced TCP Name Resolution model is a generalization and an improvement to SRM. First, port names become part of all segments exchanged during the *three-way handshake*. Second, port names have a semantic meaning depending on the domain of interpretation they belong or are intended to. Third, the TCP stack is closely integrated with DOIs for assistance in name resolution, access validation and authentication. Fourth, user processes (DOI Resolvers) implement the logic that processes *semantic* port names and decide wether the connection succeeds or not.

The modular ERM architecture allows a set of advantages over standard and SRM enabled TCP stacks:

- totaly decouples the TCP connection establishment from strict TCP port addressing;

- supports name resolution and security mechanisms, including validation, authorization and single or mutual authentication;
- supports an extensible, fault tolerant mechanism to add pluggable DOI Resolvers implementing the name resolution and connection establishment decision logic.

Similarly to SRM, services are not visible if not properly addressed (whenever name resolution or DOI Resolver's validation procedures fail). Therefore, this type of security by obscurity is also present in ERM.

The drawback of ERM, a consequence of its architecture, is the demand for a modified TCP stack on both endpoints. There is no possibility of establishing a connection between a legacy endpoint and an ERM enabled endpoint. So, this model is only applicable to DOI aware services and hosts.

Both models are transparent to applications, with the minor exception that the socket bind or connect procedures must be slightly updated. However, in SRM a legacy application can establish a connection to a named TCP service without modifications.

One problem arose during discussions detailed in Chapter 6: the fact that the destination TCP port number changed during the connection, even though it happens in the synchronization phase. Both source and destination IP addresses and TCP port numbers are used by some equipments or applications (e.g. NAT, iptables, firewalls) as a connection identifier. This problem affects either model. A discussed alternative consists on maintaining the same connection identifier philosophy in order to assure a quiet integration within already deployed network equipments.

Our prototype implementation confirmed the compatibility with other TCP implementations, validated by conformity and functional tests. Furthermore, in SRM we were able to maintain compatibility with legacy systems, kernels and applications without modifying them.

The initial proposed objectives were therefore accomplished. Two TCP name resolution models were studied, implemented and validated through tests. Backward compatibility was a concern present in each one. The architecture of the enhanced model does not enforce a strict resolution scheme neither a fixed set of allowed security algorithms. Instead, it allows a flexible integration between the TCP/IP stack and user processes.

Although TCP lacks by itself a name resolution mechanism and some access validation rules or security algorithms, we can conclude that all those features can be supported in-band during the *three-way handshake*, as proposed by ERM using its DOI Resolvers.

The developed work broadens the usage of TCP ports and allows the connection establishment to occur depending on logic which may enforce authentication, authorization or any other security action. In fact, there is now the flexibility to define different domains of interpretation, each one with its own name resolution rules and protocol.

As future work there is a possibility of writing an RFC to suggest the models here described or just parts of them, depending on further feedback from the community, specially from the "TCP Maintenance and Minor Extensions" Working Group.

More future work could also include the implementation of the suggestions discussed in Chapter 6, along with supplementary tests in heterogenous networks and between hosts deployed in the Internet. Different DOIs could be studied in detail and evaluated through the implementation of DOI Resolvers. A specific DOI could also be used to distinguish whenever the DOI logic is implemented by the final applications themselves, a way to embed the DOI Resolver functionality directly into clients and servers and to extend the address space of DOIs.

Appendices

Appendix A

Linux changes for SRM

The set of modified files within the Linux kernel source code tree and the changes made can be summarized as follows.

include/linux/in.h: definition of extended *sockaddr_in* structure;

include/linux/skbuff.h: new *skb_add_data_kernel* function that extends the current function *skb_add_data* to allow adding data available in kernel space;

include/linux/tcp.h: definition of constants for TCP socket binding and extending of structures like tcp_sock, tcp_request_sock and tcp_options_received to support port names;

include/net/checksum.h: new *csum_and_copy* function that extends the current function *csum_and_copy_from_user* to allow it to work with source address from kernel space;

include/net/sock.h: changed the *get_port* declaration in the *proto* structure to support port name;

include/net/inet_hashtables.h: major changes and enhancements including (i) new function *__inet_lookup_established* to do socket lookup based also in port name data, used by modified *inet_lookup* function (ii) new macro INET_MATCH_PORTNAME to match sockets (iii) *inet_unhash* support to deal with the port name hash (iv) new port name hash, based on the new structure *portname_hashbucket*, in order to map port names into port numbers and extension to the *inet_hashnfo* to use it;

include/net/inet_sock.h: changes to *inet_sock* structure to have port name data and some minor refinements on *inet_ehashfn* and *inet_sk_ehashfn* that use an hash (ehash) for sockets in any STATE except TCP_CLOSE and TIME_WAIT;

include/net/tcp.h: definition of constants used to set the TCP option for port name;

net/ipv4/af_inet.c: extended *inet_bind* to support bind to a local port name;

net/ipv4/inet_connection_sock.c: extended *inet_csk_get_port* to do port name lookup before obtaining a reference to local random and available port;

net/ipv4/inet_diag.c: minor fix in *inet_diag_get_exact* due to the extended *inet_lookup prototype*;

net/ipv4/inet_hashtables.c: minor change in *inet_bind_bucket_create* to save port name data, new *inet_bind_hash_portname* function that extends the functionality of the function *inet_bind_hash*, enhancements in *__inet_lookup_listener* to do socket lookup for port names, *inet_hash_connect* to hash the port name during connection process;

net/ipv4/tcp.c: changes in *tcp_init* to initialize the port name hash and enhancements to *do_tcp_setsockopt* so it supports the socket option for the binding method;

net/ipv4/tcp_input.c: extended *tcp_parse_options* to support the new TCP option, modified *tcp_rcv_synsent_state_process* to remove the socket from the established hash and add it again but indexed by the server given port instead of destination port number 0 that was used in the connection request;

net/ipv4/tcp_ipv4.c: minor changes in *tcp_v4_get_port* and *tcp_v4_err*, modified the function *tcp_v4_connect* to support new fields of *sockaddr_in* structure, enhancements to *tcp_v4_send_reset* to support the port name magic, some add-ons in *tcp_v4_conn_request* and *tcp_v4_send_synack* like recording the SYN and the SYN+ACK payload length respectively, update internal structures with port name data during *tcp_v4_syn_recv_sock*, modified *tcp_v4_do_rcv* to send the port name in the RST packet, enhanced *tcp_v4_rcv* to obtain the port name from SYN/SYN+ACK/RST packets and to validate the binding method for incoming connection requests, port name variables initialization in *tcp_v4_init_sock*;

net/ipv4/tcp_minisocks.c: update of TCP synchronization and window variables in the function *tcp_create_openreq_child* and their correct validation in *tcp_check_req*;

net/ipv4/tcp_output.c: enhancement in *tcp_syn_build_options* and *tcp_transmit_skb* to support the new TCP option, modified *tcp_make_synack* and *tcp_connect* to add the port name and update TCP synchronization and window variables in this last function.

Appendix B

Summary of Linux changes for ERM

The modifications here presented were made having the simple model as basis, thus they are incremental. The Linux kernel source code tree was modified as follows.

include/net/inet_hashtables.h: changed structure *portname_hashbucket* to use the new *resolved_portname* and *resolved_portname_len* instead of *portname* and *portname_len*; *inet_unhash* had also to be updated for that reason; macro INET_MATCH_PORTNAME, used by *__inet_lookup_established*, had to be modified to use only the port name header (*portnamehdr*) to do the socket match logic, and not the whole port name content; a similar logic was applied for looking up items inside a list in *inet_unhash*;

include/net/inet_sock.h: changes to *inet_sock* structure to store more port name variables (*resolved_portname, acked_portname, client_portname, server_portname* and respective lengths); modified *inet_ehashfn* to not use port name's length to index elements and use the port name identifier field instead, during the hash index calculation;

include/net/request_sock.h: changed the structure *request_sock_ops*, by adding support for port name parameters in the *send_reset* declaration;

include/net/sock.h: added the declaration of DOI interface functions *register_portname*, *unregister_portname*, *resolve_portname*, *validate_portname* and *revalidate_portname* to the *proto* structure;

include/net/tcp.h: definition of constant used to distinguish the different types of named segments (*NAMED_SYN, NAMED_SYNACK, NAMED_ACK, NAMED_RST*); definition of function prototypes *tcp_async_synack* and *tcp_async_ack* for dealing with asynchronous messages coming from DOI;

net/ipv4/af_inet.c: extended *inet_bind* to use *resolved_portname* and added the hook to interface DOI through *register_portname*;

net/ipv4/inet_connection_sock.c: modified *inet_csk_get_port* to use the *resolved_portname* nomenclature, while maintaining the port name→port number mapping semantics (names are compared by looking just at the *portnamehdr*);

net/ipv4/tcp_input.c: extended *tcp_parse_options* to support named ACK segments; modified *tcp_data_queue* to ignore payload on named ACK segments; implemented function *tcp_async_ack* and exported it; modified *tcp_rcv_synsent_state_process* to (i) break the processing logic there by calling *validate_portname* and (ii) to send a named ACK using *tcp_send_ack_named* on response to a named SYN+ACK and (iii) to save the resolved port name in the *reserved_portname* field of the INET socket (client side); included a reference to the external *tcp_v4_send_reset*, so it can be reused in the *tcp_async_ack* logic; minor changes in *tcp_rcv_state_process*;

net/ipv4/tcp_ipv4.c: add the DOI hook in *tcp_unhash* to invoke *unregister_portname*; enhancements in *tcp_v4_send_reset* to support an incrementing counter, per new connection, used as *id* in *portnamehdr*; updated *tcp_v4_conn_request* to the new nomenclature and to record some data in some internal variables, like the *resolved_portname* at server side; initialized some variables in *tcp_v4_syn_recv_sock*; restructured *tcp_v4_do_rcv* to deal with the different port names and to break the processing of final ACK of the *three-way handshake*, by calling a new and simplified version of *tcp_check_req* - *tcp_check_req_lite*; implemented and exported the *tcp_async_synack* function which interacts with DOI for validation of a named SYN; enhancements to *tcp_v4_rcv* including the differentiation of named segments and the asynchronous handling of SYN segments, by calling the DOI *resolve_portname* function;

net/ipv4/tcp_minisocks.c: implemented and exported the DOI *tcp_async_ackchk* function, which is used for validating the last and named ACK in the *three-way handshake*; implemented *tcp_check_req_lite*, a function similar to *tcp_check_req* but that does not create the child socket, just makes the validations and invokes the DOI *revalidate_portname* primitive; minor changes in *tcp_check_req*; added more port name related input parameters to *tcp_child_process* and inherent logic updates;

net/ipv4/tcp_output.c: implemented *tcp_send_ack_named* for sending named ACK segments (through *tcp_transmit_skb_named*); modified *tcp_send_ack* to call *tcp_send_ack_named*; extended *tcp_transmit_skb* and renamed it to *tcp_transmit_skb_named* in order to support the inclusion of the TCP port name option on-demand; made a new *tcp_transmit_skb*, which calls *tcp_transmit_skb_named* with default values, to keep semantics; some updates on the function *tcp_make_synack* due to port name nomenclature and in receive window calculation; minor changes in *tcp_connect_init* ;

Appendix C

Setting up a Linux kernel hacking environment

C.1 Kexec+Kdump and Crash

Kexec[1] is a series of patches for the Linux kernel, which provide an API, through system calls, to load a kernel from within another running kernel without rebooting the latter. There is a RPM package, *kexec-tools*, that provides the necessary tools to load the kernel. A typical usage example for kexec is depicted in Listing C.1.

```
# kexec -l /boot/bzImage --append="root=/dev/sda1"
# kexec -e
```

Listing C.1: Kexec usage example for loading the kernel image bzImage with the arguments "root=/dev/sda1"

Another tool that comes with the RPM package is *kdump*. **Kdump**[2] is a crash dumping solution which integrates closely with Kexec. When a crash occurs, the "kdump kernel" is booted, while keeping the memory preserved. It then creates a consistent dump file of the state of the previously running kernel. In */etc/kdump.conf* it's possible to specify if the dump is saved locally or even exported to another host by FTP, SCP, etc, for further analysis.

The "capture" kdump aware kernel must be loaded into a reserved memory area, that the original kernel must not use. This is configured in the boot loader. An example is shown for GRUB in Listing C.2, where the original kernel reserves 128MB of memory, starting at offset 16MB, for the loading of the kdump aware kernel.

```
title Fedora (2.6.22.9-91.portnames.fc7)
      root (hd0,0)
      kernel /vmlinuz-2.6.22.9-91.portnames.fc7 ro root=/dev/VolGroup00/LogVol00
             rhgb quiet kstack=32 crashkernel=128M@16M
      initrd /initrd-2.6.22.9-91.portnames.fc7.img
```

Listing C.2: Part of the GRUB config file, detailing

[1] http://www.xmission.com/~ebiederm/files/kexec/
http://www.ibm.com/developerworks/linux/library/l-kexec.html

[2] http://lse.sourceforge.net/kdump/

Crash[3] is a user space tool used to analyze either running Linux systems or kernel core dump files, most probably an outcome of a Linux kernel crash. By merging the functionalities of the GDB debugger, Crash features include access to the kernel log messages, to the list of the running processes, to the list of opened file descriptors and network connections among many other. One feature that is of great interest is the possibility of analyzing a kernel crash event in detail, by looking at the stack, the registers and the kernel logs. From the stack analysis, obtaining the sequence of the called functions is straightforward.

```
# crash /proc/kallsyms /usr/src/linux-2.6.22.i686/vmlinux /mnt/kdumps/vmcore
```

Listing C.3: Crash usage example (*vmlinux* is the kernel image and *vmcore* the dump)

Crash can be obtained from `http://people.redhat.com/anderson/` as a SRPM package. Configuring a system with *kexec+kdump* and *crash* simplifies kernel hacking a lot.

C.2 Ksplice

A very important contribution coming from the MIT is **Ksplice**[4]. This tool provides an hot patching mechanism for the Linux kernel itself. From the user point of view, it only requires an uncompresses kernel source tree and the source of the modified files (or (*.diff files*). The implementation is quite interesting and makes use of two kernel modules that will help during the hot update process. The tool starts by building the original kernel source tree and another one with the given patches. Then, it compares the binary object files to find out which functions were modified. Finally, it extracts the code into another object file containing the binary code that will be inserted into kernel space by the mentioned kernel modules. Ksplice works by modifying all references of the modified functions, within the live kernel, with fresh references to the updated versions of those functions. This works great if the changes are just concerned with the logic provided by the function internals. However, if changes are semantic, affecting data structures types, Ksplice cannot, at this point, proceed with the hot update.

```
user@localhost:~$ mkdir ~/linux-source/ksplice
user@localhost:~$ cp /boot/config-2.6.22 ~/linux-source/ksplice/.config
user@localhost:~$ cp /boot/System.map-2.6.22 ~/linux-source/ksplice/System.map
user@localhost:~$ cd ~/linux-source/kernel
user@localhost:~/linux-source/kernel$ cp printk.c printk.c.modified
user@localhost:~/linux-source/kernel$ ksplice-create --diffext=.modified ~/linux-source
Ksplice update tarball written to ksplice-2d3i4jgp.tar.gz
root@localhost:/home/user/linux-source/kernel# ksplice-apply ./ksplice-2d3i4jgp.tar.gz
Done!
```

Listing C.4: Ksplice usage example

Unfortunately, Ksplice only appeared during 2008 and therefore there was no time to use it within this thesis context.

[3] `http://people.redhat.com/anderson/crash_whitepaper/`
[4] `http://web.mit.edu/ksplice/`

Appendix D

Structures Definition

D.1 DOI Protocol Messages

```
#define MAX_PAYLOAD 1024  /* maximum payload size*/
#define NETLINK_TEST 17

#define DOI_MSG_TYPE_REGISTER                   0x30
#define DOI_MSG_TYPE_REGISTER_OK                0x31
#define DOI_MSG_TYPE_REGISTER_NOK               0x32
#define DOI_MSG_TYPE_UNREGISTER                 0x35
#define DOI_MSG_TYPE_UNREGISTER_OK              0x36
#define DOI_MSG_TYPE_UNREGISTER_NOK             0x37
#define DOI_MSG_TYPE_RESOLVE_REQUEST            0x38
#define DOI_MSG_TYPE_RESOLVE_ANSWER             0x39
#define DOI_MSG_TYPE_REGISTER_PORTNAME          0x40
#define DOI_MSG_TYPE_REGISTER_PORTNAME_OK       0x41
#define DOI_MSG_TYPE_REGISTER_PORTNAME_NOK      0x42
#define DOI_MSG_TYPE_UNREGISTER_PORTNAME        0x43
#define DOI_MSG_TYPE_UNREGISTER_PORTNAME_OK     0x44
#define DOI_MSG_TYPE_UNREGISTER_PORTNAME_NOK    0x45
#define DOI_MSG_TYPE_VALIDATE_REQUEST           0x46
#define DOI_MSG_TYPE_VALIDATE_ANSWER            0x47
#define DOI_MSG_TYPE_REVALIDATE_REQUEST         0x48
#define DOI_MSG_TYPE_REVALIDATE_ANSWER          0x49


struct doi_msg_register_portname{
        unsigned short int      resolved_port;
        unsigned short int      resolved_portname_len;
} doi_msg_register_portname;

struct doi_msg_resolve_request{
        unsigned short int      portname_len;
} doi_msg_resolve_request;

struct doi_msg_resolve_answer{
        unsigned short int      resolved_port;
        unsigned short int      resolved_portname_len;
        void                    *resolved_portname;
        unsigned short int      server_portname_len;
        void                    *server_portname;
        unsigned char           result;
} doi_msg_resolve_answer;

struct doi_msg_validate_request{
        struct sock             *sk;
        struct sk_buff          *skb;
        unsigned short int      portname_len;
```

```
        unsigned short int      resolved_portname_len;
} doi_msg_validate_request;

struct doi_msg_validate_answer{
        struct sock             *sk;
        struct sk_buff          *skb;
        unsigned short int      acked_portname_len;
        void                    *acked_portname;
        unsigned short int      client_portname_len;
        void                    *client_portname;
        unsigned char           result;
} doi_msg_validate_answer;

struct doi_msg_revalidate_request{
        struct sock             *sk;
        struct sk_buff          *skb;
        struct request_sock     *req;
        struct request_sock     *prev;
        unsigned short int      server_portname_len;
        unsigned short int      acked_portname_len;
} doi_msg_revalidate_request;

struct doi_msg_revalidate_answer{
        struct sock             *sk;
        struct sk_buff          *skb;
        struct request_sock     *req;
        struct request_sock     *prev;
        unsigned char           result;
} doi_msg_revalidate_answer;
```

Appendix E

Function Prototypes

E.1 DOI Protocol API - User Processes (*DOI resolvers*)

```
/* send a DOI message, through a Netlink socket, with the given DOI payload data */
 void send_doi_msg(int sock_fd,unsigned int pid, unsigned int id,unsigned char doi,
              unsigned char doi_msghdr_type, unsigned short doi_msghdr_payload_len,
              void *payload);

/* receive a DOI message, from a Netlink socket, with the given flags for recvmsg() */
 int receive_doi_msg_flags(int sock_fd,struct msghdr *msg,int flags);

/* receive a DOI message, from a Netlink socket, with no specific recvmsg() flags */
 int receive_doi_msg(int sock_fd,struct msghdr *msg);
```

E.2 DOI Protocol API - Kernel (*DOI interface LKM*)

```
/* send a DOI message, through a Netlink socket, with the given DOI payload data */
 void send_doi_msg(struct sock *nl_sk, unsigned int pid, unsigned int id,
              unsigned char doi_msghdr_type, unsigned short doi_msghdr_payload_len,
              void *payload);

/* send a DOI message, through a Netlink socket, with the given DOI payload data */
 void send_doi_msg_with_header(struct sock *nl_sk, unsigned int pid, unsigned int id,
              unsigned char doi_msghdr_type, unsigned short hdr_payload_len,
              void *hdr_payload,unsigned int payload_len, void *payload);

/* send a DOI message, through a Netlink socket, with the two given DOI payloads */
 void send_doi_msg_with_header2(struct sock *nl_sk, unsigned int pid, unsigned int id,
              unsigned char doi_msghdr_type, unsigned short hdr_payload_len,
              void *hdr_payload, unsigned int payload1_len, void *payload1,
              unsigned int payload2_len,void *payload2);

/* receive a DOI message, from a Netlink socket */
 void receive_doi_msg(struct sock *nl_sk, struct sk_buff **skb, int *err);
```

Bibliography

[1] M. Allman, S. Floyd, and C. Partridge. Increasing TCP's Initial Window. RFC 3390, IETF, October 2002.

[2] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. RFC 2581, IETF, April 1999.

[3] P. Barham, S. Hand, R. Isaacs, P. Jardetzky, R. Mortier, and T. Roscoe. Techniques for Lightweight Concealment and Authentication in IP Networks. Technical Report IRB-TR-02-009, Intel Research Berkeley, 2002.

[4] R. Braden. Requirements for Internet Hosts – Communication Layers. RFC 1122, IETF, October 1989.

[5] R. Braden. Extending TCP for Transactions – Concepts. RFC 1379, IETF, November 1992.

[6] R. Braden. T/TCP – TCP Extensions for Transactions Functional Specification. RFC 1644, IETF, July 1994.

[7] D.D. Clark. Window and Acknowledgement Strategy in TCP. RFC 813, IETF, July 1982.

[8] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol, Version 1.1. RFC 4346, IETF, April 2006.

[9] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[10] M. Duke, R. Braden, W. Eddy, and E. Blanton. A Roadmap for Transmission Control Protocol (TCP) Specification Documents. RFC 4614, IETF, September 2006.

[11] W. Eddy. TCP SYN Flooding Attacks and Common Mitigations. RFC 4987, IETF, August 2007.

[12] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, IETF, May 1994.

[13] Sérgio Freire and André Zúquete. A TCP-layer name service for TCP ports. *USENIX Annual Technical Conference*, pages 275–286, 2008.

[14] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782, IETF, February 2000.

[15] K. Harrenstien, M.K. Stahl, and E.J. Feinler. DoD Internet host table specification. RFC 952, IETF, October 1985.

[16] A. Heffernan. Protection of BGP Sessions via the TCP MD5 Signature Option. RFC 2385, IETF, August 1998.

[17] ISO. ISO Transport Protocol specification ISO DP 8073. RFC 905, IETF, April 1984.

[18] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, IETF, May 1992.

[19] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, IETF, December 2005.

[20] M. Krzywinski. Port Knocking: Network Authentication Across Closed Ports. *SysAdmin Magazine*, (12):12–17, 2003.

[21] M. Lottor. TCP port service Multiplexer (TCPMUX). RFC 1078, IETF, November 1988.

[22] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and application protocol scrubbing. In *INFOCOM (3)*, pages 1381–1390, 2000.

[23] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018, IETF, October 1996.

[24] George H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

[25] P.V. Mockapetris. Domain names – concepts and facilities. RFC 1034, IETF, November 1987.

[26] P.V. Mockapetris. Domain names – implementation and specification. RFC 1035, IETF, November 1987.

[27] J. Postel. Internet Protocol. RFC 791, IETF, September 1981.

[28] J. Postel. Transmission Control Protocol. RFC 793, IETF, September 1981.

[29] J. Postel. TCP maximum segment size and related topics. RFC 879, IETF, November 1983.

[30] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[31] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, IETF, September 2001.

[32] Michael Rash. Single packet authorization. *Linux J.*, 2007(156):1, 2007.

[33] J. Reynolds. Assigned Numbers: RFC 1700 is Replaced by an On-line Database. RFC 3232, IETF, January 2002.

[34] J. Reynolds and J. Postel. Assigned Numbers. RFC 1700, IETF, October 1994.

[35] J. Salim, H. Khosravi, A. Kleen, and A. Kuznetsov. Linux Netlink as an IP Services Protocol. RFC 3549, IETF, July 2003.

[36] J. Sermersheim. Lightweight Directory Access Protocol (LDAP): The Protocol. RFC 4511, IETF, June 2006.

[37] R. Srinivasan. Binding Protocols for ONC RPC Version 2. RFC 1833, IETF, August 1995.

[38] R. Srinivasan. RPC: Remote Procedure Call Protocol Specification Version 2. RFC 1831, IETF, August 1995.

[39] P. Srisuresh and M. Holdrege. IP Network Address Translator (NAT) Terminology and Considerations. RFC 2663, IETF, August 1999.

[40] S. Yamaguchi T. Tsutsumi. Secure TCP — providing security functions in TCP layer. *Proc. INET 95 - Internet Society's 1995 International Networking Conference*, 1995.

[41] J. Touch. A TCP Option for Port Names. Internet draft (expired), IETF, April 2006.

[42] W. Yeong, T. Howes, and S. Kille. X.500 Lightweight Directory Access Protocol. RFC 1487, IETF, July 1993.

[43] André Zúquete. *Segurança em Redes Informáticas.* FCA, 2nd, edition, 2008.

Index

U

V